VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Shlukování na základě hustoty pro velká data

Density Based Classification for Big Data

2018

Vojtěch Bill

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Vojtěch Bill**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Shlukování na základě hustoty pro velká data
Density Based Classification for Big Data

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem projektu je implementovat několik metod pro shlukování dat na základě měření hustoty jejich rozložení a upravit je pro efektivní běh v paralelním prostředí nad velkými daty. Paralelizace bude řešena pomocí OpenMP, MPI nebo CUDA. Výsledné algoritmy budou otestovány nad dodanými daty a výsledky analýzy budou přehledně zobrazeny a vyhodnoceny. Implementace bude v jazyce C++.

Práce bude obsahovat:
1. Seznámení s problematikou.
2. Aktuální State of the art.
3. Podrobný popis zvoleného/zvolených algoritmů.
4. Experimenty, měření, vyhodnocení.
5. Závěr - zhodnocení výsledků.

Seznam doporučené odborné literatury:

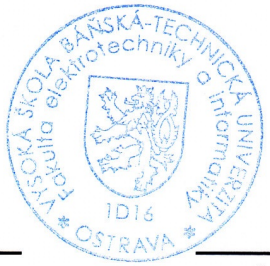
[1] Mohammed J. Zaki, Wagner Meira, Jr., Data Mining and Analysis: Fundamental Concepts and Algorithms, Cambridge University Press, May 2014. ISBN: 9780521766333.
[2] Aggarwal C.C. (2015), Data Mining: The Textbook, Springer.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Jan Platoš, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018



doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 27. dubna 2018 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 27. dubna 2018

.....

Rád bych na tomto místě poděkoval všem, kteří mi s vypracováním pomohli, zejména pak vedoucímu práce doc. Ing. Janu Platošovi, Ph.D. za odborné konzultace a rady při řešení problémů, na které jsem při zpracovávání této práce narazil.

Abstrakt

Tato diplomová práce se věnujeme problematice shlukování, zejména pak shlukování na základě hustoty s důrazem na běh pro velká data. V první části práce jsou popsány teoretické poznatky, věnujeme se shlukování podle hustoty rozložení dat v prostoru a velká část je věnována popisu algoritmu DBSCAN. Následně se seznámíme s pokročilými datovými strukturami pro ukládání dotazovaných dat. Ve druhé části implementujeme vlastní řešení upraveného algoritmu DBSCAN, který využívá kd-strom jako datovou strukturu a umožňuje paralelní běh. V závěru změříme výsledky paralelizovaného řešení v porovnání se sekvenčním algoritmem, porovnáme kd-strom s přístupem dotazování hrubou silou a podíváme se na možné návrhy dalšího vylepšení.

Klíčová slova: shlukování, DBSCAN, datová struktura, k-d strom, paralelizace, OpenMP

Abstract

This diploma thesis focuses on clustering with special interest in density based cluster analysis for big data. In the beginnig, there is a theory behind clustering and mainly behind density based cluster analysis and the DBSCAN algorithm. Significant part of the first half of this theses consists of the data structures for efficient data storage and quering. In the second part, we propose our own version of DBSCAN with kd-tree used as a data structure and with parallel aproach of some of DBSCAN's steps. We than measure the impact of parallelizing the DBSCAN algorithm and compare the basic approach of querying data using brute force in contrast to kd-tree. In the final part we propose possible enhancements and functionality for further improvement.

Key Words: clustering, DBSCAN, data structure, k-d tree, parallelization, OpenMP

Obsah

Seznam použitých zkratek a symbolů

DBSCAN	–	Density-Based Spatial Clustering of Applications with Noise
k-d strom	–	k-dimenzionální strom
NN	–	Nearest Neighbor
ANN	–	Approximate Nearest Neighbor
OMP	–	Open Multi-Processing
MPI	–	Message Passing Interface
CUDA	–	Compute Unified Device Architecture
LSH	–	Locality-Sensitive Hashing

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Shlukování patří mezi nejznámější způsoby klasifikace dat. Jeho použití je běžné v řadě oblastí lidské činnosti a jeho univerzálnost a snadná představivost umožňují, že jej velmi často používají i lidé mimo matematické nebo technické zaměření. V této práci se postupně seznámíme s konceptem shlukování, rozdělíme si jej na hierarchické a nehierarchické a každou z částí si popíšeme detailně. Budeme se věnovat převážně shlukování na základě hustoty a algoritmu DBSCAN. Se shlukováním souvisí také uložení a dotazovaní zpracovávaných dat, podíváme se proto na pokročilé datové struktury k tomu určené. Mezi tyto struktury patří například kvadratický strom, oktalový strom a kd-strom. U těchto struktur si popíšeme způsob, jakým data rozdělují, jejich výhody a nevýhody a možné použití.

S použitím knihovny, která implementuje kd-strom si v druhé polovině práce vytvoříme vlastní verzi algoritmu DBSCAN. Popíšeme si samotnou knihovnu, způsob práce algoritmu DBSCAN a jeho modifikaci, která jej připraví pro paralelizaci. Následně se pokusíme za použití technologie OpenMP tento algoritmu paralelizovat pro běh na více vláknech. Naše řešení změříme a zanalyzujeme výsledky. Následně se podíváme na další možnosti rozšíření a popíšeme jiná publikovaná řešení.

2 Shluková analýza

Shluková analýza je statistická metoda klasifikace dat. Jejím cílem je přiřazovat jednotlivé reprezentanty dat do shluků, na základě jejich podobnosti. Výsledkem shlukové analýzy je určitý počet shluků, které obsahují podobná data. Zatímco podobnost jednotlivých prvků v rámci shluku by měla být v ideálním případě co nejvyšší, prvky z ostatních shluků by pak měli být naopak co nejvíce odlišné.

Shlukovou analýzu dělíme na hierarchickou a nehierarchickou. Jednotlivé algoritmy shlukové analýzy se liší podle počtu a typu vstupních parametrů, podle toho, zda umí detekovat odlehlé prvky, podle nutnosti předem určovat či neurčovat počet shluků apod. Další způsob dělení je dělení shlukovacích algoritmů na tzv. *hard* a *soft*, u *hard* shlukování přiřadíme každý bod pouze do jednoho shluku. Opakem je pak shlukování typu *soft*, kde je každému objektu vypočtena pravděpodobnost s jakou náleží tomu konkrétnímu shluku, pravděpodobnost je poté rozdělena mezi všechny shluky tak, aby součet byl vždy 1. U klasického, pevného (*hard*), shlukování je konkrétní objekt přiřazen do konkrétního shluku tedy vždy s pravděpodobností 1 nebo 0 (náleží nebo nenáleží).

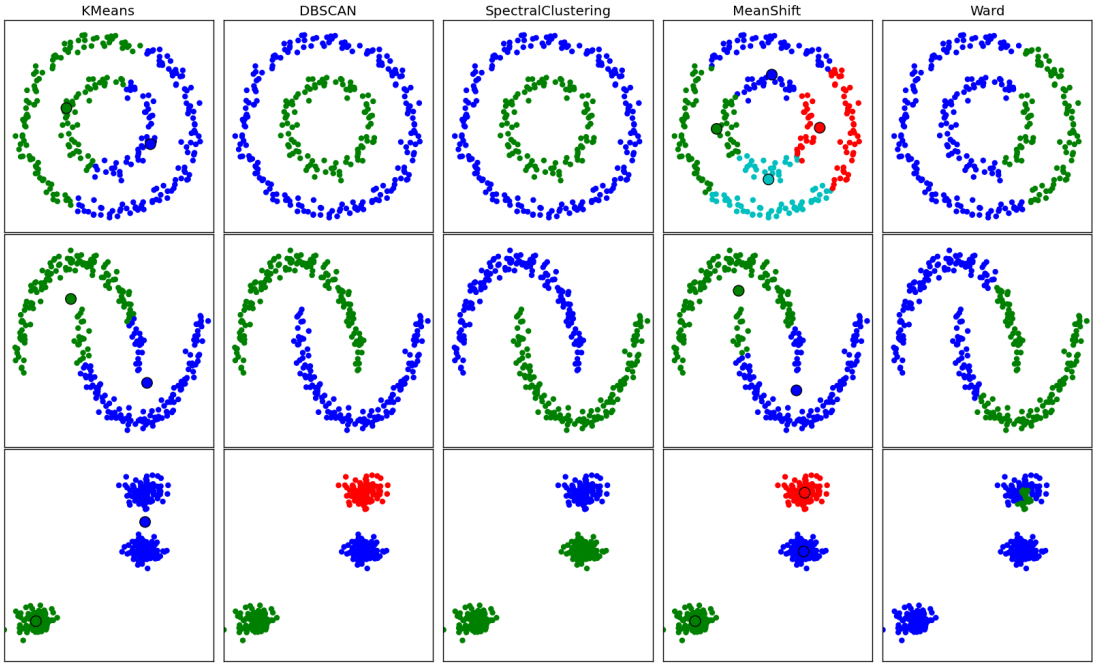
Využití shlukovacích algoritmů je velmi široké a obecně můžeme říct, že se hodí pro všechny případy, ve kterých je žádoucí rozdělit data do co nejvíce homogenních skupin. Lehce představitelné je například dělení zákazníků internetových obchodů pro marketingové účely, finanční analýza rizikovosti zákazníků, práce s naměřenými biologickými daty apod. Pro jednotlivé úkoly pak volíme nejlépe odpovídající algoritmy, kde hraje roli například počet shluků (algoritmy s pevně zadaným počtem), schopností detekce šumu (noise reduction, detekce outlierů) apod. Porovnání toho, jak si se stejnými daty poradí rozdílné shlukovací algoritmy si můžeme prohlédnout na obrázku 1.

2.1 Hierarchická shluková analýza

Hierarchické shlukování je v podstatě posloupnost rozkladů, kdy máme na jedné straně shluk obsahující všechna pozorování a na druhé straně pak shluky obsahující jeden prvek. Na základě toho můžeme říct, že průnikem dvou shluků může být buď prázdná množina, nebo jeden z těchto shluků. Na hierarchické shlukování lze nahlížet ze dvou stran – divizní přístup, kdy vycházíme z celku, obsahující všechna pozorování a ten následně dělíme na menší shluky a přístup aglomerativní, kdy naopak každé pozorování je v samostatném shluku a ty pak spojujeme. Postup shlukování a výsledek hierarchické shlukové metody nejčastěji znázorňujeme pomocí dendrogramu, který přehledně zobrazuje shluky a jejich vztahy navzájem.

2.1.1 Aglomerativní hierarchické shlukování

U tohoto druhu hierarchického shlukování začínáme s jednoprvkovými shluky a postupně je spojujeme na základě podobnosti jednotlivých měření až dokud nemáme jediný shluk, případně



Obrázek 1: Porovnání rozdělení dat do shluků pomocí odlišných shlukovacích algoritmů[1]

nedosáhneme předem daného počtu shluků [1].

Obecný postup při aglomerativním hierarchickém shlukování je:

1. Vypočteme si matici vzdáleností mezi jednotlivými pozorováními. Tato matice pak označuje jak jsou si dva zvolené objekty blízké (podobné).
2. Začneme od n shluků, kdy každý shluk obsahuje jedno pozorování.
3. V matici vzdáleností najdeme dva nejbližší objekty.
4. Spojíme tyto dva objekty do jednoho shluku a v matici vzdálenosti vymažeme řádky a sloupce pro vybrané objekty a vytvoříme nový řádek a sloupec pro nový shluk. Vzdálenosti od tohoto shluku počítáme podle vybrané metody aglomerativního shlukování.
5. Zaznamenáme si pořadí cyklu, spojené shluky a úroveň spojení.
6. Pokud jsme nespojily všechny objekty do jednoho shluku, pokračujeme znovu bodem 3.

Mezi metody výpočtu vzdálenosti shluků patří například:

- Metoda nejbližšího souseda: U této metody se bere nejbližší objekt, nenáležící do aktuálního shluku.

[1]Obrázek převzat z: Comparing different clustering algorithms on toy datasets. *scikit-learn* [online]. [cit. 22.4.2018]. Dostupné z: `http://ogrisel.github.io/scikit-learn.org/sklearn-tutorial/auto_examples/cluster/plot_cluster_comparison.html`

- Metoda nejvzdálenějšího souseda: Zde se naopak berou v potaz nejvzdálenější objekty porovnávaných shluků. Odpadá zde problém řetězení z metody nejbližšího souseda.
- Centroidová metoda: Při výpočtu vzdálenosti mezi shluky vycházíme ze vzdálenosti centroidů jednotlivých shluků. Nevýhoda je v případě spojování dvou shluků velmi rozdílné velikosti, kdy centroid nového shluku bude velmi blízko původnímu centroidu většího ze shluků.
- Wardova metoda: Je podobná metodě centroidové a její výhoda spočívá v tendenci tvořit shluky podobné velikosti.

Rozdíly vypočítaných shluků za použití jednotlivých metod si můžeme prohlédnout na obrázku 2.

Obrázek 2: Shluky vytvořené za pomocí tří odlišných metod.

2.1.2 Divizní hierarchické shlukování

U divizního přístupu postupujeme opačně než u postupu aglomerativního. Na začátku máme množinu všech objektů, které dále dělíme na menší shluky a tímto postupem vytváříme hierarchický systém. Jednotlivé shluky v krocích postupně dělíme na dva nové dle zvoleného kritéria. Končíme v situaci, kdy máme všechny shluky jednoprvkové. Nevýhodou tohoto postupu je jeho exponenciální časová náročnost při nalézání vhodného rozdělení množiny na dvě podmnožiny. Z tohoto důvodu se používají divizní metody pouze pro malý počet objektů [2].

2.2 Nehierarchická shluková analýza

U tohoto přístupu se shlukováním nevytváří hierarchická struktura a jednotlivé shluky mezi sebou nemají žádný vztah. Úkolem je opět rozdělit množinu objektů do podmnožin podle předem zadaných kritérii. U nehierarchického shlukování nedochází po rozdělení do shluků k další úrovní rozdělování a je tedy většinou snaha o rozdělení dat do shluků tak, aby byly objekty v rámci jednoho shluku co nejhomogennější, objekty ze shluků dalších naopak co nejvíce odlišné a částečně také aby jednotlivé shluky byly velikostně podobné, což ovšem závisí na druhu dat. U některých metod máme jako vstupní kritérium udán počet shluků, případně zkoušíme různá nastavení algoritmu a podle výsledku se rozhodujeme, které nastavení nám dodává nejlepší výsledky.

2.2.1 k-means

K-means je jedním z nejznámějších shlukovacích algoritmů vůbec. Je to zástupce nehierarchického shlukovaní, nevytváří tedy mezi shluky žádný vztah. U tohoto algoritmu je potřeba předem zadat počet shluků, případně je možné zkusit více různých vstupních nastavení a porovnat výsledky. K-means pracuje na principu centroidů, kterými je definován každý shluk. Objekt se zařazuje do nejbližšího centroidu, přičemž míra blízkosti se počítá jednou ze zvolených metrik. Algoritmus postupuje iterativně, takže po každém průchodu daty se přepočítají polohy centroidů. Algoritmus je velmi závislý na volbě počtu shluků a na počáteční poloze jednotlivých centroidů. Často se tedy postupuje tak, že necháme algoritmus pracovat několikrát, pokaždé s jiným počátečním nastavením a následně vybereme nejlepší výsledek.

Postup k-means algoritmu je zobrazen na obrázku 3 a je následující:

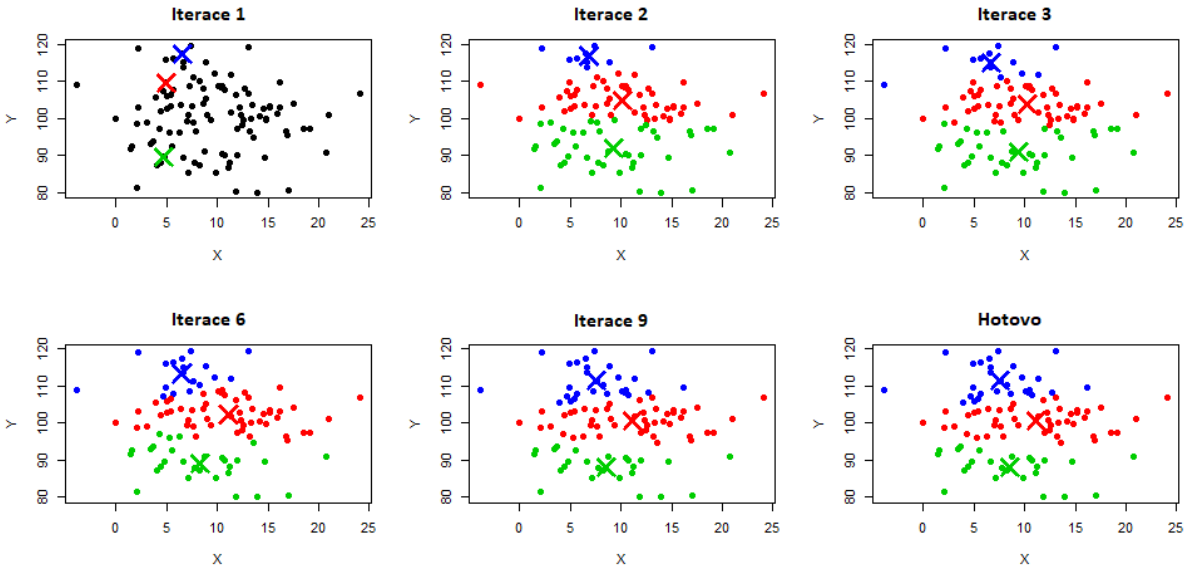
1. k centroidů se náhodně vybere z datasetu, těmito body jsou pak definovány počáteční polohy centroidů.
2. Pro každý objekt z datasetu se najde nejbližší centroid podle zvolené metriky a objekt se přiřadí do daného shluku.
3. Pro každý shluk se přepočtou metody centroidů.
4. Pokud zůstala po přepočtení poloha centroidů stejná, algoritmus končí. V opačném případě pokračujeme opět bodem 2.

Algoritmus konverguje většinou poměrně rychle, ovšem v závislosti na počátečních centroidech a počtu shluků nám nemusí dodávat kvalitní výsledky. Proto je obvykle vhodné použít jednu z metod hodnocení kvality shlukování a porovnat několik běhů algoritmu s náhodně vygenerovanými vstupy.

2.2.2 Fuzzy shlukování

Hlavním rozdílem u fuzzy shlukování oproti předchozím metodám je možnost přiřadit jeden objekt do více shluků. U každého pozorování zde vypočítáváme pravděpodobnost, že bude příslušet do daného shluku, tzv. koeficient příslušnosti. Můžeme tedy říct, že u všech předchozích metod byla pravděpodobnost přiřazení do shluku buď 1 nebo 0. U fuzzy metody je tomu jinak a pravděpodobnost může nabývat hodnot i mezi těmito mezníky. Obecně lze říct, že u fuzzy shlukování je přítomnost objektu rozdělena do všech shluků. Algoritmus pracuje podobně jako k-means.

1. Vybere se počet shluků
2. Každému objektu se náhodně přiřadí koeficient příslušnosti
3. Pro každý shluk se přepočte střed.



Obrázek 3: Postupné iterace algoritmu k-means.[2]

4. Pro každý objekt se spočítá pravděpodobnost ke všem shlukům.

5. Pokud se koeficient příslušnosti nezměnil méně, než je práh citlivosti, opakuj bod 3.

2.3 Shlukování na základě hustoty

Shlukování dat podle hustoty jejich distribuce (anglicky: density based clustering) je jeden z přístupů ve shlukové analýze. Jak napovídá název, data se zde primárně přiřazují do shluků podle toho v jakém jsou okolí. Výhodou tohoto přístupu je to, že dokáže přiřadit do jednoho shluku i pozorování, které jsou od sebe velmi vzdáleny, ale patří do společné struktury. Příklad takto distribuovaných dat si můžeme prohlédnout na obrázku 7. Shluky tvoří objekty hustě osídlené v určitých regionech v prostoru. Jednotlivé shluky odděluje řídce osídlená oblast. V okolí shluků a mezi nimi se mohou nacházet odlehlá pozorování (*noise*), některé algoritmy je umí detekovat a označit.

Základní předpoklady těchto algoritmů jsou:

1. Shluky jsou hustě osídlené oblasti v prostoru. Jsou odděleny oblastmi s nízkou distribucí prvků.

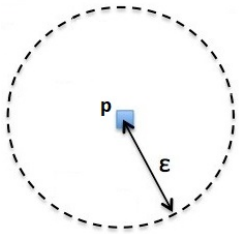
2. Shluk je maximální množina vzájemně dosažitelných prvků.

[2]Obrázek převzat z: K-Means Clustering – What it is and How it Works. *Learn by Marketing* [online]. [cit. 22.4.2018]. Dostupné z: `http://www.learnbymarketing.com/methods/k-means-clustering/`

3. Shluky mohou být libovolných tvarů, mohou se lišit ve velikosti nebo může jeden shluk obalovat jiný.

ϵ (*epsilon*) je vzdálenost, která určuje sousedství objektu. Jestliže ve vzdálenosti $<= \epsilon$ od objektu leží jiný objekt, patří do jeho sousedství (anglicky používaný výraz ϵ-neighborhood).

Vysoká hustota – sousedství daného objektu obsahuje alespoň minPts objektů. Sousedství bodu ohraničené vzdáleností *epsilon* je vidět na obrázku 4.



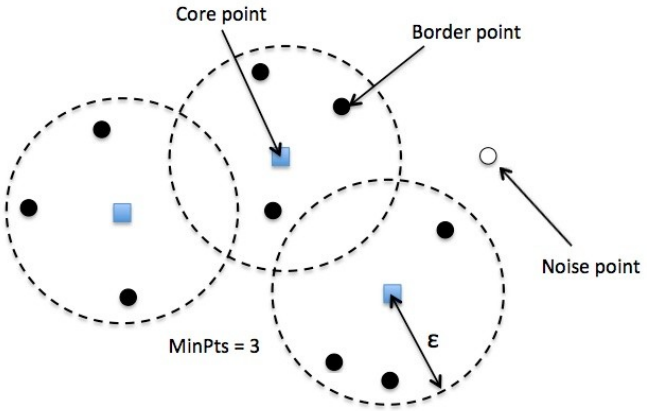
Obrázek 4: Bod p a jeho ϵ-sousedství.

Pomocí ϵ a minPts následně rozdělujeme objekty do tří skupin.

Core point je objekt, který má více než minPts ve svém sousedství epsilon. Tyto objekty tvoří vnitřní část shluku.

Border point neboli okrajový bod je objekt, který má ve svém ϵ sousedství méně než minPts objektů, ale je v sousedství *core pointu*.

Noise point neboli šum je objekt, který není ani *core* bodem ani *border* bodem. Grafické znázornění vidíme na obrázku 5



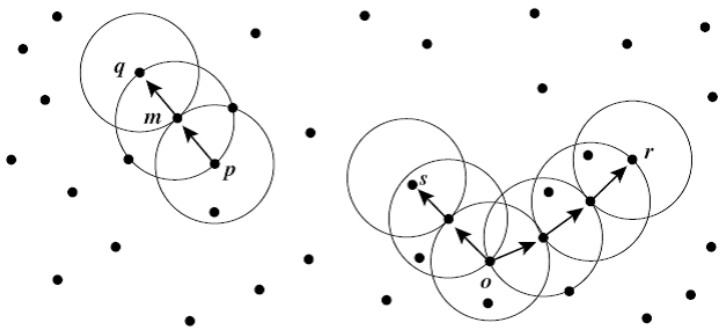
Obrázek 5: Rozdělení bodů na core, border a noise při nastavení minPts = 3

Všechny objekty v datasetu musí po provedení shlukování patřit do jedné ze skupin. *Core* a *border* body tvoří samotné shluky a *noise*, neboli šum jsou pak body v meziprostoru.

Dosažitelnost bodů v datasetu (anglicky density-reachability) dělíme na přímou a nepřímou. Dosažitelnost nám říká, jestli lze na dva dané objekty dosáhnout (patří do jednoho shluku), nebo nikoliv.

Objekt q je přímo dosažitelný z objektu p, jestliže p je *core* bod a objekt q leží v jeho ϵ-sousedství. V tomto případě platí, že pokud je p *core* bod a q nikoliv, tak q je přímo dosažitelný z p, p není přímo dosažitelný z q. Dosažitelnost je tedy asymetrická relace.

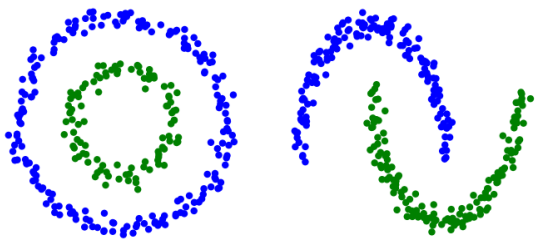
Nepřímá dosažitelnost nám poté určuje, zda jsme schopni dosáhnout i na vzdálenější objekty v datasetu pomocí řetězce několika objektů. Bod p je přímo dosažitelný z $p2$, $p2$ je přímo dosažitelný z bodu $p1$ a $p1$ je přímo dosažitelný z bodu q. Dostaneme tedy řetězec bodů $p < p2 < p1 < q$. V tomto případě bude bod p nepřímo dosažitelný z bodu q. Relace je opět asymetrická, bod q nemusí být přímo dosažitelný z bodu p (není-li p *core* bod) 6.



Obrázek 6: Přímá dosažitelnost bodů q-m a m-p (vlevo) a nepřímá dosažitelnost bodů s-r (vpravo).

2.4 DBSCAN

Typický představitel shlukování dle hustoty je DBSCAN. Jeho název je tvořen počátečními písmeny anglických slov *density-based spatial clustering of applications with noise* a česky by se dal přeložit jako shlukování na základě hustoty s detekcí šumu. Jeho počátek sahá do roku 1996 kdy jej představili autoři Martin Ester, Hans-Peter Kriegel, Jörg Sander a Xiaowei Xu. Algoritmus je schopen seskupit do shluku objekty v hustě osídlených oblastech (objekty s mnoha blízkými sousedy) a označit jako šum body, které leží v oblastech s řidším osídlením. Díky tomu je schopen detekovat shluky, které nejsou některými jinými algoritmy odlišitelné, například shluk uvnitř jiného shluku, jak je vidět na obrázku 7 [3].



Obrázek 7: Data jsou zařazena do shluků podle hustoty distribuce.

DBSCAN rozděluje body do třech skupin:

1. Bod p je *core* bod, pokud má alespoň minimální počet bodů ve svém sousedství ve vzdáleností menší než ϵ.

2. Bod q je *border* bod, pokud je přímo dosažitelný z bodu p (pokud leží maximálně ve vzdálenosti ϵ) a bod p je *core* bod. Zároveň ale ve svém ϵ sousedství nemá dostatečný počet bodů, aby se stal *core* bodem.

3. Body, které jsou ve vzdálenosti delší, než ϵ od *core* bodu jsou označeny jako šum.

Pokud je bod p *core* bod, tvoří shluk společně se všemi dalšími body, které jsou z něj přímo či nepřímo dosažitelné. Každý shluk musí obsahovat alespoň jeden *core* bod. Body, které nejsou *core* můžou být součástí shluku, ale tvoří jeho okraj, protože nemohou být použity pro další rozšiřování.

Dosažitelnost (anglicky density-reachability) není symetrická relace, protože žádný bod není dosažitelný z bodu, který není *core* bodem. Propojitelnost (anglicky density-connectedness) naopak je symetrická relace a určuje, že body p a q jsou propojitelné, existuje-li bod o, ze kterého jsou oba body dosažitelné. Shluk splňuje u algoritmu DBSCAN následující dvě podmínky:

1. Všechny objekty ve shluku jsou propojitelné.

2. Všechny objekty dosažitelné z jakéhokoliv bodu ve shluku jsou součástí shluku.

Jak bylo popsáno v předchozí kapitole, DBSCAN potřebuje pro svou práci nastavené dvou počátečních parametrů: ϵ pro nastavení velikosti rádiusu ohraničujícího sousedství objektu a minPts jako minimální počet objektů v sousedství, které jsou požadovány, aby vytvořily oblast se zvýšenou hustotou.

Algoritmus začíná návštěvou libovolného bodu. Pokud tento bod obsahuje ve svém ϵ sousedství obsahuje potřebný počet bodů (minPts), vytvoříme nový shluk. V ostatních případech je bod označen jako šum. Je dobré si uvědomit, že tento bod může být později označen jako sousední bod jiného bodu, který jej přiřadí do svého shluku.

Pokud daný bod obsahuje ve svém sousedství dostatek bodů a tvoří tedy základ shluku, všechny body v jeho sousedství jsou také přiřazeny do daného shluku. Stejný postup uplatníme pro tyto body v sousedství, pokud splňují dané podmínky. Takto pokračujeme až dokud nenajdeme celý shluk. Jakmile je shluk dokončen, vyzkoušíme libovolný jiný nenavštívený bod a aplikujeme stejný algoritmus, vedoucí k založení nového shluku.

```
DBSCAN(DB, dist, eps, minPts) {
   C = 0                                          // pocitadlo shluku
   for each bod P in dataset DB {
      if druh(P) != neprirazen then continue       // vem novy nenavstiveny bod
      Sousedstvi N = RangeQuery(DB, dist, P, eps) // najdi sousedy
      if |N| < minPts then {                      // kontrola hustoty sousedstvi
```

```
            druh(P) = Sum                              // oznac jako sum
            continue
        }
        C = C + 1                                      // novy shluk
        druh(P) = C                                    // novy pocatecni bod
        Seed set S = N \ {P}                           // sousedstvi pro expanzi
        for each bod Q in S {                          // zpracuj vsechny body v
            sousedstvi
            if druh(Q) = Sum then druh(Q) = C          // zmena sumu na border
            if druh(Q) != neprirazen then continue     // jiz navstiven
            druh(Q) = C                                // oznac souseda
            Sousedstvi N = RangeQuery(DB, dist, Q, eps) // najdi sousedy
            if |N| >= minPts then {                    // kontrola hustoty
                S = S + N                              // pridej nove sousedy do
                    mnoziny
            }
        }
    }
}
```

Výpis 1: Algoritmus DBSCAN zapsaný v pseudokódu

```
RangeQuery(DB, dist, Q, eps) {
    Sousedstvi = empty list
    for each bod P in database DB {                // projdi vsechny body v
        datasetu
        if dist(Q, P) <= eps then {                // spocti vzdalenost a
            porovnej s eps
            Sousedstvi = Sousedstvi + {P}          // pridej do promenne
                vysledky
        }
    }
    return Neighbors
}
```

Výpis 2: Funkce RangeQuery zapsaná v pseudokódu

Mezi výhody algoritmu DBSCAN řadíme:

- Na začátku není potřeba algoritmu udávat přesný počet shluků.

- Tvar shluku nehraje roli, pomocí algoritmu můžeme najít i shluk, který obalen jiným shlukem. Pokud nejsou spojeny (záleží na nastavení parametrů minPts a ϵ).
- Detekce a označení šumu.

Mezi nevýhody pak řadíme:

- Může se stát, že *border* bod na hranici dvou shluků bude patřit do obou shluků (v závislosti na seřazení dat nebo pořadí vyhodnocování bodů). Tato situace nemůže z podstaty algoritmu nastat pro body typu *core* a *noise*.
- Výsledná kvalita shlukování velmi závisí na nastavení obou parametrů minPts a ϵ. Velice závisí na datech (jejich distribuci, počtu dimenzí a způsobu výpočtu vzdálenosti mezi objekty), při nevhodném nastavení počátečních parametrů můžeme dostat výsledky s žádnou vypovídací hodnotou.
- Pokud jsou v datasetu velké rozdíly v hustotě dat, nemusí být možné najít rozumný kompromis pro nastavení parametrů minPts a ϵ tak, aby správně fungoval pro všechny shluky.

Z popisu algoritmu je poměrně jasné, že klíčové pro to, aby správně pracoval, je počáteční nastavení jeho dvou parametrů. Při správném nastavení můžeme očekávat kvalitní rozdělení datasetu do shluků, avšak špatné nastavené, byť i jednoho z parametrů může mít ve výsledku velmi špatný efekt na výslednou podobu shluků. DBSCAN je citlivý na oba své parametry, obecně lze říct, že ϵ se nastavuje podle druhu řešeného problému (vzdálenost objektů od sebe) a minPts pak určuje hlavně velikost shluků. Nastavení se ale liší případ od případu a pro jeho správnost je klíčový přehled o analyzovaných datech.

minPts: základním pravidlem bývá, že minPts lze bez podrobného přehledu o datech nastavit s ohledem na počet dimenzí D, obvykle se doporučuje nastavení podle pravidla minPts >= D + 1. Minimální hodnota, která dává smysl je 3, jelikož při nastavení minPts = 1 je jasné, že každý bod by při tomto nastavení byl shlukem sám o sobě. Nastavení vyšší hodnoty je obvykle vhodnější z hlediska odrušení šumu a výsledkem budou silnější shluky.

ϵ: pokud zvolíme hodnotu ϵ příliš malou, velká část dat může zůstat nezařazena do shluků. Příliš velká hodnota pak inklinuje ke sjednocení shluků do jednoho velkého shluku a zařazení většiny objektů do tohoto obřího shluku. Obecně se doporučují spíš menší hodnoty vedoucí ke konkrétnějším a menším shlukům.

Aplikaci algoritmu DBSCAN, jako jednoho z hojně používaných algoritmů, najdeme například ve:

- Apache Commonst Math - obsahuje implementaci DBSCANu v jazyce Java
- R - v balíčku DBSCAN nalezneme implementaci v C++ s použitím kd-stromu
- scikit-learn - implementace v Pythonu a také v C++, s Euklidovskou metrikou vzdálenosti
- Weka - obsahuje implementaci algoritmu s kd-stromem

2.5 Metriky výpočtu vzdálenosti

Při zpracování shlukové analýzy potřebujeme velmi často zjistit vzdálenost jednotlivých pozorování. K tomuto účelu se používají metriky, pomocí kterých určujeme vzdálenost dvou pozorování. Každá z metrik se liší způsobem, jakým výslednou vzdálenost počítá. Podle druhu dat, případně podle zvoleného algoritmu se poté používá nejvíce vyhovující metrika.

Eukleidovská metrika:

$$d(\mathbf{p},\mathbf{q}) = \sqrt{(p_1-q_1)^2+(p_2-q_2)^2+\cdots+(p_i-q_i)^2+\cdots+(p_n-q_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2} \quad (1)$$

Manhattanská metrika:

$$d(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\|_1 = \sum_{i=1}^{n}|p_i-q_i| \quad (2)$$

Čebyševova metrika:

$$d(\mathbf{p},\mathbf{q}) = \max_i(|p_i-q_i|) \quad (3)$$

2.6 Silhouette koeficient

Při porovnávání výsledků shlukovacích algoritmů vyvstává otázka, jak objektivně měřit kvalitu shlukování. Silhouette koeficient kombinuje navzájem soudržnost (kohezi) prvků v rámci shluku a separaci prvků mezi jednotlivými shluky. Kvalita shlukování měřena tímto koeficientem tedy informuje, jak moc jsou prvky ve shluku podobné sobě navzájem a odlišné od prvků v jiných shlucích. Jako měřítko podobnosti zde používáme metriku vzdálenosti jednotlivých prvků od sebe. Z toho pak také plyne, že měření kvality shlukování tímto způsobem se nám hodí pro určitý typ shlukovacích algoritmů. Zejména pro algoritmy typu k-Means, kde nám jde o primárně o podobnost prvků na základě vzdálenosti. Méně vhodný je pak pro shlukování, kde nejde primárně o vzdálenost všech prvků, ale například jejich distribuci v prostoru. Následující tři kroky popisují, jak vypočítáme koeficient pro jeden prvek:

1. Pro objekt i vypočítej průměrnou vzdálenost ke všem ostatním objektům ve svém shluku, označíme ji a(i).

2. Pro objekt i a každý shluk, který jej neobsahuje vypočítej průměrnou vzdálenost i od každého bodu v daném shluku. Najdi nejnižší hodnotu ze všech shluků, označíme si ji b(i).

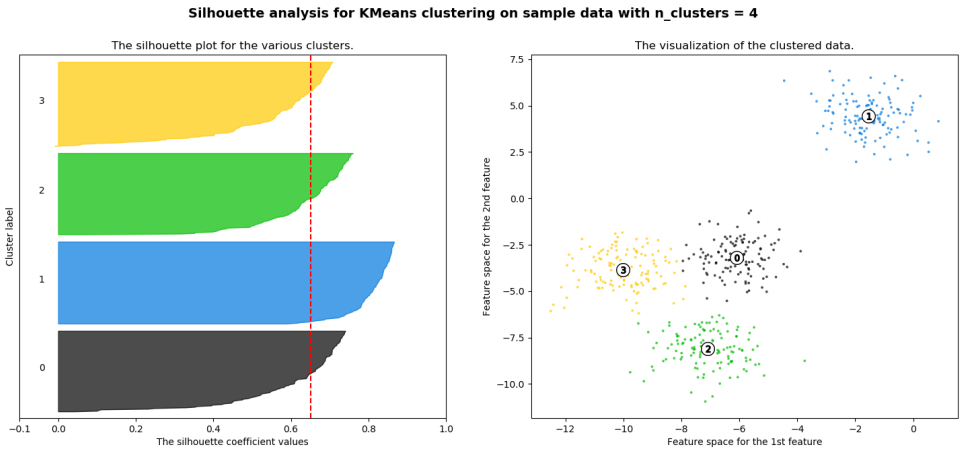
3. Vypočítej hodnotu silhouette koeficientu pro prvek i, vzorec pro výpočet je:

$$s(i) = \frac{b(i)-a(i)}{\max\{a(i),b(i)\}} \quad (4)$$

Hodnota koeficientu může nabývat hodnot v rozmezí od -1 po 1. Záporná hodnota nám říká, že bod má blíž k bodům jiného shluku než k bodů v rámci svého shluku, takováto hodnota je tedy pro nás většinou nežádoucí. Snahou je docílit kladné hodnoty koeficientu (a(i) < b(i)), protože v takové situaci mají prvky v jednom shluku podobnější vlastnosti, než k nejpodobnějšímu z ostatních shluků.

Výsledný průměrný silhouette koeficient pro jeden shluk poté můžeme vypočítat tak, že zprůměrujeme jednotlivé hodnoty koeficientu bodů v daném shluku. Celkový koeficient také spočítáme zprůměrováním všech koeficientů jednotlivých prvků v datasetu [4].

Graficky můžeme zobrazit výsledek Silhouette analýzy v grafu, který nám přehledně ukazuje kvalitu rozdělení dat do shluků. Na obrázku 8 lze přehledně vidět výsledek shlukování algoritmu k-Means se čtyřmi shluky. Podle velikosti jednotlivých sloupců v grafu vidíme, že každý shluk je přibližně stejně veliký. Podle hodnoty koeficientu na ose x pak vidíme, že žádný ze shluků se nevymyká tím, že by měl výrazně nižší hodnoty koeficientu než shluky ostatní. V tomto případě jsou tedy průměry silhouette koeficientu kladné a rozložené rovnoměrně kolem celkového průměru všech bodů.



Obrázek 8: Grafická reprezentace Silhouette analýzy pro algoritmus k-Means se čtyřmi shluky.[3]

[3]Obrázek převzat z: Selecting the number of clusters with silhouette analysis on KMeans clustering. *scikit-learn* [online]. [cit. 22.4.2018]. Dostupné z: `http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html`

3 Pokročilé datové struktury pro ukládání dat

Datová struktura je programová konstrukce, která slouží pro efektivní správu dat v počítačovém programu. Díky vhodné volbě a implementaci datové struktury můžeme výrazně zefektivnit běh algoritmu nebo i celého programu, který s daty pracuje. Data jsou v počítači reprezentována pomocí různých datových typů. Tyto typy pak můžeme dále skládat do struktur, díky kterým s daty můžeme pracovat snáze a rychleji.

Datové struktury zařazujeme mezi abstraktní datové typy, nezajímá nás jejich konkrétní implementace, v jakém programovacím jazyce jsou napsány ani konkrétní typ ukládaných dat. Rozdíl mezi jednotlivými datovými strukturami je uspořádání jejich vnitřní struktury a operace, které je možné nad takto uloženými daty provádět.

Dnes jsou známy desítky datových struktur, přičemž každá je vhodná pro řešení jiného problému. Z hlediska datové struktury a vhodnosti jejího použití nás zajímá především její paměťová náročnost, operace, které nad daty je schopna poskytnout a také jejich asymptotická složitost. S operacemi nad daty se zde setkáváme velmi často a jejich základní přehled by se dal shrnout do několika oblastní: efektivní ukládání, vyhledávání, třídění nebo řazení a mazání. Existují struktury, které zvládnou velmi rychle určitý druh operací, a naopak pokulhávají s jinými a naopak. Pro každou operaci jde vypočítat asymptotická složitost, na základě které můžeme porovnat vhodnost použití struktury pro řešení určeného problému.

Základní dělení struktur podle jejich složitosti bychom mohli popsat takto: Základní takové struktury:

- Pole (statická a dynamická) - jedná se o uspořádanou množinu objektů stejného datového typu. Ukládání probíhá v navazujících prvcích za sebe a přístup k prvkům je řešen pomocí indexu. Můžeme používat jedno i více rozměrná pole. U dvou a více rozměrných polí přistupujeme k prvkům podobně jako například do tabulky, s použitím více indexů.

- Spojový seznam (jednosměrně zřetězený, obousměrně zřetězený a kruhový) - jeho hlavní výhodou je, že umožňuje efektivně ukládat data předem neznámé délky. Data jsou uloženy ve formě uzlů, kde každý uzel obsahuje samotný datový objekt a také ukazatel na jeden nebo více dalších uzlů.

- Fronta (princip FIFO - First-In-First-Out) - vhodná pro ukládání proměnlivého počtu datových objektů s principem FIFO. Fronta pracuje pouze s prvním a posledním prvkem. K tomuto používá tři základní operace: zařazení prvku, vyřazení prvku a test prázdnoty (enqueue, dequeue, empty).

- Zásobník (princip LIFO Last-In-First-Out) - na rozdíl od fronty se zde objekty ukládají a vytahují na principu LIFO, poslední zařazený prvek je tedy vyřazen jako první. Zásobník pracuje pouze s posledním vloženým prvkem. Základní operace jsou vložit, vybrat a test prázdnoty (push, pop, empty).

Pokročilé datové struktury:

- Stromy - Ve stromech řadíme data od kořene pomocí větvení vnitřními uzly až k listům. Data jsou řazena hierarchicky a při dotazování vždy porovnáváme klíč v uzlu a eliminujeme podstatnou část dat, takže výsledné vyhledávání ve stromové struktuře je velmi rychlé. Existují spousty modifikací stromů. Podle větvení (binární, B-strom, halda), zda jsou vyvážené či nikoliv apod.

- Grafy - datové struktury, které jsou složeny z uzlů a hran. Datové objekty jsou uchovávány v uzlech a hrany spojují uzly a tím vytvářejí vzájemné vazby. Základní dělení můžeme popsat jako grafy neorientované, u kterých nerozlišujeme směr hran a grafy orientované, u kterých je naopak určen počáteční a koncový uzel.

- Hashe - jedná se o strukturu, které výrazně snižuje složitost vyhledávání pomocí indexování. Při volbě vhodné hashovací funkce můžeme za pomocí klíče indexovat data a sestavit tak tabulku, na které jsme následně schopni vyhledávat až s logaritmickou složitostí.

3.1 Kvadratický strom (quadtree)

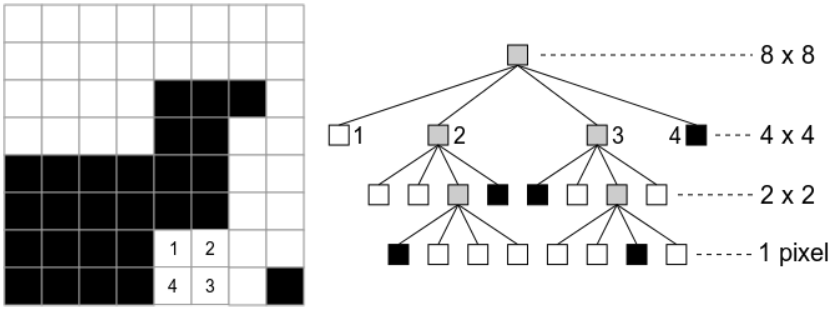
Kvadratický strom je stromová datová struktura, u které má každý vnitřní uzel přesně čtyři potomky. Jedná se tedy v podstatě o dvoudimenzionální analogii oktalových stromů. Díky této své vlastnosti se kvadratický strom hojně používá pro práci s dvoudimenzionálním prostorem například obrazovým materiálem. Kvadratický strom je používán zejména na dělení dvoudimenzionálních prostorů rekurzivně tak, že daný prostor vždy rozdělí na čtyři podprostory. Listové uzly jsou většinou asociovány s danými daty a reprezentují tak v podstatě objekt zájmu. Z tohoto hlediska lze použít kvadratický strom jako velmi efektivní datovou strukturu pro shlukovací algoritmy v dvoudimenzionálním prostoru.

Princip dělení prostoru pomocí kvadratického stromu funguje ve čtyřech základních krocích:

1. Rozděl daný dvoudimenzionální prostor do čtyř částí.
2. Pokud prostor obsahuje jeden nebo více bodů vytvoř potomka a ulož jej do větve daného prostoru.
3. Pokud prostor neobsahuje ani jeden bod potomka nevytvářej.
4. Rekurzivně pokračuj bodem č. 1 pro každého potomka.

Jednoduchou reprezentaci kvadratického stromu nad rastrovaným prostorem si můžeme prohlédnout na obrázku 9.

Při dotazovaní se nad kvadratickým stromem začínáme u kořene stromu, prozkoumáme každý z potomků a zkontrolujeme, zda protíná oblast, kam se dotazujeme. Pokud zjistíme, že ano, rekurzivně se zanoříme do této větve stromu. Pokud narazíme na listový uzel, zjistíme zda obsahuje nějaký bod, a ten případně vrátíme. Díky rekurzivnímu rozdělení prostoru je dotazování



Obrázek 9: Kvadratický strom vytvořený nad rastrem o velikosti 8x8.[4]

se nad stromem velmi rychlé. V případě, že chceme najít nejbližšího souseda pro daný bod odpadá nutnost procházet všechny body v datasetu, každým krokem totiž velkou část dat eliminujeme a procházíme pouze nejbližší kvadranty k danému bodu.

Existuje více variant kvadratických stromů, liší se například v tom, jakým způsobem přistupují k dělení prostoru. Podprostory můžou nabývat čtvercových i obdélníkových tvarů. Další modifikace může rozdělovat prostor tak, ať každý kvadrant obsahuje maximálně jeden bod (obrázek 10) nebo také bodů několik. V případě, že chceme mít například v každém kvadrantu omezený počet bodů, lze strom konstruovat tak, že v případě vyššího počtu bodů v kvadrantu se tento bude dělit až do doby dosažení požadovaného prahu bodů na kvadrant.

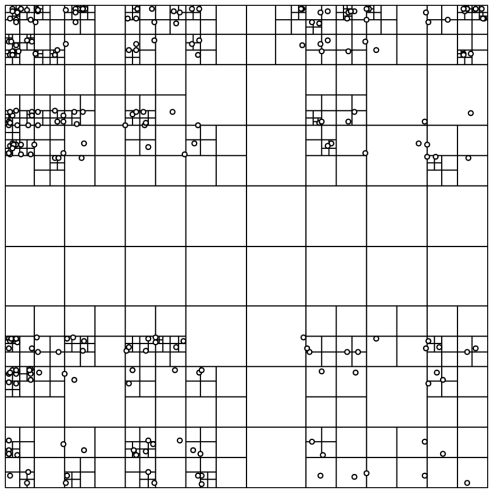
Použití kvadratických stromů odpovídá jejich zaměření na právě dvě dimenze, nejčastěji se s ním setkáme v oblastech jako je reprezentace obrazových dat, zpracování obrazu, indexování geografických dat, detekce kolizí ve dvou dimenzionálních prostorech (typicky v počítačových hrách), ukládání řídce distribuovaných dat (například informace o formátování dokumentů na počítači), hra života a analýza fraktálových obrazců.

3.2 Oktalový strom (octree)

U oktalového stromu platí velmi podobná analogie, jakou jsme si popsali v kapitole o kvadratických stromem ovšem s rozdílem, že místo dvoudimenzionálního prostoru zde pracujeme s dimenzemi třemi. Každý vnitřní uzel stromu má tedy přesně osm potomků. Můžeme si jej představit jako dělení krychle na polovinu v každé ze třech dimenzí. Oktalový strom dělí rekurzivně třídimenzionální prostor do osmi podprostorů.

Z hlediska analýzy dat v prostoru je jeho použití vhodné u případů, které jsou reprezentovány pomocí klasických třech dimenzí. Většina vlastností oktalového stromu se shoduje s vlastnostmi stromu kvadratického. Opět zde vycházíme z kořenového uzlu, ten má ovšem u oktalového stromu přesně osm potomků. Situace se rekurzivně opakuje až dokud nenarazíme na listy stromu. Díky

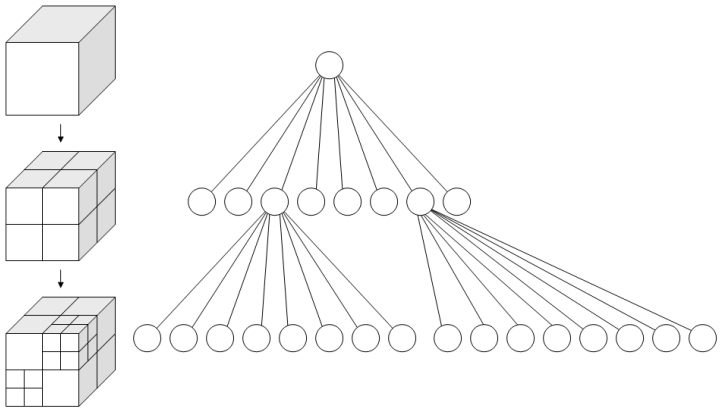
[4]Obrázky převzaty z: Quadtree. *Wikipedia* [online]. [cit. 22.4.2018]. Dostupné z: `https://en.wikipedia.org/wiki/Quadtree`



Obrázek 10: Rozdělení datasetu v 2D prostoru pomocí kvadratického stromu tak, aby každý kvadrant obsahoval maximálně jeden bod.

jedné dimenzi navíc se počet uzlů ve stromu podstatně rychleji zvyšuje s tím, jak stoupá počet úrovní.

Použití oktalového stromu je opět nejčastější v počítačové grafice, detekci kolizí v trojdimenzionálním prostoru, ale také díky právě třem dimenzím se používá pro zpracování barevného spektra RGB a redukci barev.



Obrázek 11: Princip dělení 3D prostoru a konstrukce oktaloveho stromu.

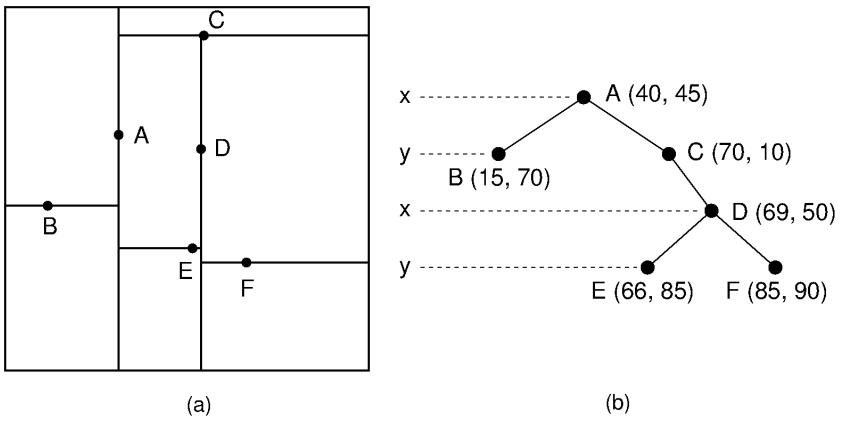
3.3 kd-strom

Prohledávání určitého prostoru má většinou vysokou výpočetní složitost, protože s narůstajícími body v prostoru narůstá nelineárně časová a paměťová náročnost celé operace. Tento proces lze zefektivnit tím, že místo celého prostoru budeme prohledávat pouze určité jeho části, čímž velmi výrazně zredukujeme celkovou složitost a zrychlíme běh programu. Jako příklad řešení tohoto problému jsme si v kapitolách 3.1 a 3.2 ukázali použití dvou datových struktur. Jejich hlavními

nevýhodami je přesně určený počet dimenzí prostoru, se kterým pracujeme. Obecnějším řešením, které není závislé ja počtu dimenzí je použití kd-stromů. kd-strom (od slova k-dimensionální) je datová struktura, která nám umožňuje vymezit datový prostor pomocí polorovin na podčásti, které obsahují přímo jeden bod v prostoru anebo podstatně redukují prohledávanou množinu bodů, takže jejich procházení je časově i paměťově významně úspornější.

kd-strom prostor rozděluje postupně tak, že jednotlivé osy dělí nadrovinou kolmou k dané ose. Bod, ve kterém je osa rozdělena je mediánem souřadnic bodů v daném podintervalu, který vznikl dělením předchozího intervalu (obrázek 12). Tímto postupem se postupně dopracujeme až k situaci, kdy je strom vyvážený a rozděluje prostor až do takové míry, že každý podprostor obsahuje pouze 1 bod. V takovéto struktuře je následně vyhledávání nejbližšího souseda podstatně rychlejší, stačí nám postupně procházet pouze ty podprostory, kde se daný bod nachází, což zjistíme ze souřadnic jednotlivých dimenzí.

Existuje mnoho modifikací kd-stromu, které se liší například jiným přístupem k dělení os. kd-strom je speciálním případem binárního rozdělování prostoru pomocí binárního stromu, jedná se v podstatě o multidimenzionální binární rozdělování v prostoru. Každý uzel v kd-stromu je k-dimenzionální bod, každý uzel, který není listem, může být považován za jakousi rozdělující rovinu, která dělí prostor na dvě poloviny. Body, které jsou v levé části jsou ve stromu reprezentovány větví ležící nalevo, body napravo pak tvoří pravou část podstromu. Směr této rozdělující roviny volíme tímto způsobem: každý uzel ve stromu je asociován s jednou z k dimenzí, přičemž rovina protíná kolmo právě osu této dimenze. Pokud například dělíme osu x, všechny body v podstromu, které mají menší hodnotu x, než tento uzel se objeví v levé větvi a analogicky všechny body mající hodnotu x větší napravo [5].



Obrázek 12: 2D prostor rozdělený kd-stromem, samotný strom je vidět vpravo.[5]

[5]Obrázek převzat z: KD Trees. *OpenDSA* [online]. [cit. 22.4.2018]. Dostupné z: `https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/KDtree.html`

3.3.1 Tvorba stromu

Jelikož existuje mnoho způsobů, jak lze dělit jednotlivé osy nadrovinou, existuje také mnoho přístupu ke konstrukci kd-stromů. Metoda, která vede k vyváženému kd-stromu splňuje následující podmínky:

1. Body se do stromu vkládají za použití mediánového bodu, ze všech bodů právě vkládaných do dané větve stromu.

2. Když se zanořujeme do stromu, procházíme postupně osy, které byly použity pro dělení nadrovinami. U 3D prostoru má například kořen stromu osu dělenou v dimenzi x, oba jeho potomci pak v ose y, jejich potomci v ose z. Celý proces se opět opakuje.

Není nutné vždy volit mediánový bod, nicméně pak není zaručeno, že bude strom vyvážený. Protože hledání mediánu ve velkém množství bodů může být časově náročná operace, lze použít například přístup náhodného výběru několika bodů z datasetu a vypočítat medián z těchto bodů. Tento přístup nezaručuje vyvážený strom, ale vede k výsledkům, které k němu mají velmi blízko.

Dalším způsobem, jak lze konstruovat vyvážený kd-strom je ten, že si před samotnou konstrukcí stromu setřídíme body a vyhneme se tak problému hledání mediánu pro každou dimenzi.

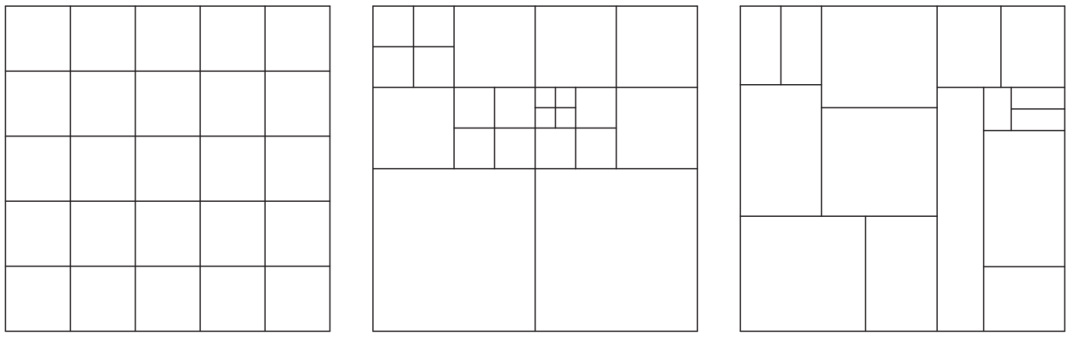
Přidávání bodů do stromu probíhá stejným způsobem jako v každém jiném vyhledávacím stromu. Začneme procházet strom od kořene a zanořujeme se do levé nebo pravé větve stromu, podle toho, zda bod náleží do levé nebo pravé strany dělené dimenze. Jakmile dojdeme až do uzlu, pod kterým by měl nový bod ležet, přidáme nový bod jako levý nebo pravý list daného uzlu, opět podle toho, do jaké části dělené dimenze spadá. Tímto způsobem se může strom dostat z vyváženého stavu a celková efektivita práce s ním se tím sníží. Jak výrazně poklesne efektivita vyhledávání ve stromu záleží na distribuci vkládaných bodů v prostoru a také jak velkou část výsledného stromu tvoří nově vložené body. V případě výrazné nevyváženosti stromu je možné jej znovu vyvážit a tím obnovit efektivitu práce s ním.

Odebírání bodů ze stromu lze řešit například tak, že najdeme uzel, který obsahuje odebíraný bod a znovu vytvoříme celou část stromu pod ním. Druhým způsobem by bylo nalezení náhradního uzlu za ten, který mažeme. Náhradní uzel najdeme v podstromu, jehož kořenem je odebíraný uzel. Ten nahradíme novým bodem a takto rekurzivně pokračujeme v podstromu. Pokud je odebíraný uzel zároveň listem, není třeba hledat náhradu.

3.3.2 Hledání nejbližšího souseda (nearest neighbour search)

Hledání nejbližšího souseda (NN search) je algoritmus, jehož úkolem je nalézt nejbližší bod k danému vstupnímu bodu. Díky zkonstruovanému stromu je toto hledání velmi efektivní a využívá naplno vlastnosti stromu. Díky tomu, že jsme schopni eliminovat velké množství prohledávaného prostoru nemusíme procházet všechny body z datasetu.

Hledání nejbližšího souseda probíhá v několika krocích:



Obrázek 13: Rozdělení 2D prostoru pomocí rovnoměrné mřížky (vlevo), kvadratickým stromem (vprostřed) a pomocí kd-stromu (vpravo).

1. Začínáme u kořene stromu a rekurzivně procházíme stromem stejně, jako bychom vkládali nová data. Zanoříme se vlevo nebo vpravo v závislosti na tom, zda je bod menší nebo větší než rozdělení v prohledávané dimenzi.

2. Jakmile dorazíme až k listu, uložíme si konkrétní uzel jako aktuální nejlepší výsledek.

3. Vynořujeme se postupně z rekurze a na každém uzlu provádíme následné kroky:

   (a) Pokud je aktuální uzel blíž ke vstupnímu bodu než aktuální nejlepší výsledek, uložíme si tento uzel jako aktuální nejlepší výsledek.

   (b) Zkontrolujeme, jestli může být na druhé straně rozdělené dimenze bližší bod, než je náš aktuální nejlepší výsledek. V podstatě nám stačí projít okolí vstupního bodu v rádiusu rovnému vzdálenosti k aktuálnímu nejlepšímu výsledku. Jelikož jsou všechny roviny, které rozdělují dimenze zarovnané s osami, stačí nám jednoduché porovnání koordinát, abychom zjistili, zda je rozdělení dimenze blíže ke vstupnímu bodu, než aktuální nejlepší výsledek. V tomto případě je možné, že je za rozdělením dimenze bližší bod, musíme se tedy zanořit do druhé větve stromu a porovnat vzdálenosti. Pokud je rozdělení dimenze dále, není třeba prostor na druhé straně prohledávat a algoritmus stoupá vzhůru ke kořenu stromu.

4. Když dorazíme ke kořeni, algoritmus je u konce. Aktuální nejlepší výsledek se stává konečným a našli jsme nejbližší bod k bodu vstupnímu.

Pro větší rychlost výpočtu je vhodné použít jako metriku výpočtu vzdálenosti méně výpočetně náročné kalkulace a také si ukládat hodnoty nejlepšího výsledku, aby je algoritmus nepočítal při každém porovnání.

Obecně platí, že při velikém počtu dimenzí musí algoritmus navštěvovat výrazně více větví stromu a je tedy pomalejší než u prostorů s nízkým počtem dimenzí. Také pokud je počet bodů jen nepatrně větší, než počet dimenzí, není algoritmus o mnoho rychlejší, než lineární prohledávání kompletního datasetu. Modifikace tohoto algoritmu mohou hledat také více než

jeden nejbližší bod a ukládat k nejbližších bodů, při tomto způsobu práce se eliminuje větev stromu jen pokud nemůže nabídnou bližší bod, než je k aktuálních nejbližších bodů. Díky tomu, že algoritmus rozděluje rozsah dimenze na polovinu v každé úrovní stromu, je také vhodný pro hledání dat v rozsahu určité dimenze.

4 Knihovna ANN

ANN označuje první písmena anglických slov Approximate Nearest Neighbors (přibližní nejbližší sousedé) a tvoří také název knihovny pro práci s kd-stromy s použitím vyhledávání Approximate Nearest Neighbors a Nearest Neighbors. Autory jsou David M. Mount a Sunil Arya, její aktuální verze (1.1.2.) byla uvolněna v roce 2010.

ANN je napsána v jazyce C++ a je to knihovna určená pro akademické testování pokročilých datových struktur a práci s vyhledáváním v mnoha dimenzionálních datech. Její činnost spočívá v tom, že z vložených bodů v multidimenzionálním prostoru vybuduje datovou strukturu. Následně se na tuto strukturu dotazujeme tak, že vytvoříme dotaz na bod q a výsledkem bude jeden, případně k bodů, nejbližších našemu bodu q. Díky použití pokročilých struktur pro ukládání dat je tento dotaz velmi rychle zpracován. Vzdálenost mezi body lze počítat mnoha známými metrikami (včetně euklidovské, manhattanské a dalších). Podle autorů knihovny má tato velmi dobré výsledky v datech, kde je počet dimenzí do 20. Součástí celého balíku je také testovací program, přeložená knihovna ve formě `.dll` souboru a program pro vizualizaci struktury [6].

4.1 Popis knihovny

Problém hledání nejbližšího souseda v setu dat spočívá v zadání setu datových bodů P v d-dimenzionálním prostoru. Tyto body jsou předzpracovány do datové struktury, takže v případě dotazu na bod q jsme schopni vrátit nejbližší (případně k nejbližších) bodů P, které leží nejblíže bodu q. Body jsou uloženy jako vektory souřadnic, případně integer. Vzdálenost mezi dvěma body můžeme definovat několika způsoby, lze použít všechny klasické metriky měření vzdálenosti.

Efektivní řešení problému nejbližšího souseda, zvláště pro větší počet dimenzí, je poměrně náročný problém. Nejsnáze lze postupovat metodou hrubé síly, tedy pro každý dotazovaný bod se spočítají vzdálenosti ke všem bodům v datasetu a jejich porovnáním, případně seřazením vzdáleností získáme odpověď. Tento způsob řešení je ale časově a výpočetně velmi náročný a obzvlášť pro aplikace s velkým počtem dotazů je pro větší množství dat prakticky nepoužitelný. Efektivnějším způsobem je tedy vytvořit si předzpracovanou strukturu, ze které následně zjistíme při každém jediném dotazu rychle odpověď.

Jeden z největších problémů je, že prakticky všechny metody kromě řešení hrubou silou rostou velmi strmě z hlediska časové i paměťové náročnosti vzhledem k vzrůstajícímu počtu dimenzí. Při velkém počtu dimenzí, řádově desítky dimenzí, je pak podle autorů knihovny velmi malá rozdíl oproti klasickému přístupu hrubou silou. Knihovna implementuje také metodu tzv. přibližného nejbližšího souseda (approximate nearest neighbour), která zachovává obrovskou rychlost zpracování dotazu i při velikém počtu dimenzí, přičemž vrácený bod (případně k bodů), nemusí být nutně nejbližší, ale rozdíl ve vzdálenosti není významný.

Knihovna nabízí následující funkcionalitu a vlastnosti:

ANN, Version 1.1
Copyright, 2010 David M. Mount

Obrázek 14: logo knihovny ANN pro práci s pokročilými datovými strukturami

- k - nearest neighbor hledání. Nastavením parametru k určujeme, kolik nejbližších bodů k hledanému bodu dostaneme jako výsledek operace hledání nejbližšího souseda. Metoda se tedy neomezuje pouze na nalezení jednoho nejbližšího bodu, ale podporuje hledání libovolně nastavitelného počtu bodů, které leží v nejmenší vzdálenosti od našeho bodu.

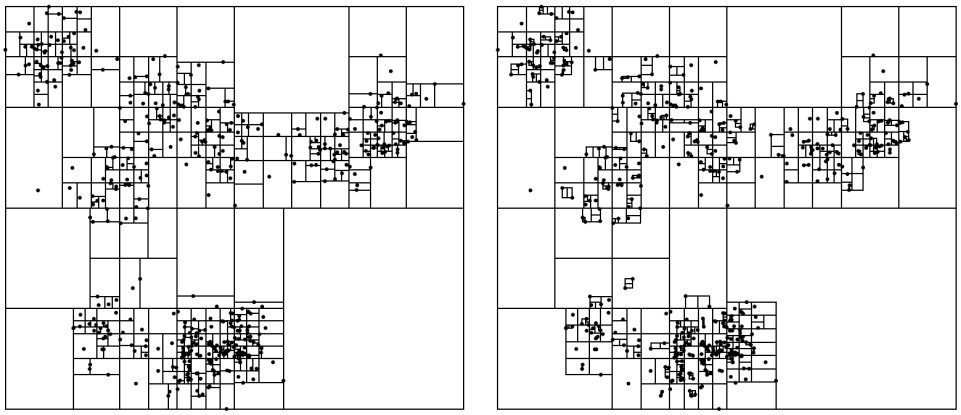
- nearest neighbor i approximate nearest neighbor hledání. První metoda nalezne nejbližší bod/y k hledanému bodu, bez jakékoliv tolerance chyby. Druhá metoda poté nabízí nalezení nejbližšího bodu/ů s drobnou tolerancí chyby. Nemusí tedy vrátit úplně nejbližší bod, ale výsledný rozdíl ve vzdálenosti nejbližšího bodu od bodu vráceného bude velmi malý (a lze nastavit parametrem chybovosti *epsilon*), rychlost hledání v tomto režimu je pak obecně rychlejší. Obětujeme tedy přesnost za rychlost a tato funkcionalita si u některých algoritmů může najít své opodstatnění.

- běžně používané metriky počítání vzdáleností

- časová a prostorová náročnost je lineární vzhledem k počtu bodů n a počtu dimenzí d a není závislá na dalších parametrech. Výsledná předzpracovaná datová struktura je ve výsledku o něco větší než zpracovávaný dataset a její velikost se s přibývajícími daty zvětšuje lineárně.

Původním záměrem knihovny je testování algoritmů hledání nejbližšího souseda specificky založených na rozkládání prohledávaného prostoru. Svým zaměřením se tedy velmi hodí jako pokročilá datová struktura pro algoritmy s častým dotazováním na okolní body, například DBSCAN. Mimo ukládání dat do struktury k-d tree je také součástí knihovny datová struktura box-decomposition tree. Srovnání toho, jakým způsobem obě struktury rozdělí 2D prostor nad stejným datasetem si můžeme prohlédnout na obrázku 15.

Mimo klasické vytvoření a dotazování se nad vytvořenou strukturou jsou ke knihovně k dispozici dva testovací programy pro testování a vyhodnocování metod.



Obrázek 15: Rozdělení dvoudimenzionálního prostoru pro k-d tree a box-decomnposition tree. Obrázek je výstupem programu `ann2fig`, který je součástí knihovny.

4.2 Použití

Pro kompilaci je možné použít standardní ANSI C++ překladač. Úspěšně ji lze kompilovat na systémech Unix, Linux, Solaris, Linux Red Hat 2.X a dalších. Pro naši práci je důležitá podpora operačního systému Windows a prostředí a překladače Microsoft Visual Studio.

Na systému Windows lze s knihovnou pracovat dvěma způsoby. Prvním z nich je samotné sestavení projektu Ann.sln v režimu "Release". Po kompilaci najdeme ve složce projektu v podsložkách soubory `ANN.lib` a `ANN.dll`, které následně potřebujeme mít přiloženy ve svém projektu, používajícím knihovnu ANN. Druhým způsobem, který využijí primárně lidé, kteří nepotřebují měnit nic ve zdrojovém kódů programu, je využití již připravených a přeložených souborů, které autoři dávají v dispozici mezi soubory projektu. V tomto případě jsou již všechny potřebné části přeloženy pro práci a stačí je připojit do používaného projektu.

Pro to, abychom mohli využít funkce obsažené v knihovně, musíme mít ve zdrojovém kódu include hlavičkového souboru. Tento header soubor knihovny ANN obsahuje deklarace objektů a procedur knihovny. Odkaz na hlavičkový soubor zajistíme standardním přidáním řádku do programu:

```
#include <ANN/ANN.h>
```

Každý bod v d-dimenzionálním prostoru je v knihovně reprezentován jako vektor velikosti d a typu `ANNcoord`. Pro účely porovnání vzdáleností mezi body není v základním nastavení ANN potřeba pokaždé počítat odmocninu jako v případě pravé Euklidovské metriky, ale počítá se místo ní tzv. *squared distance*, tedy rozdíl vzdáleností mezi danou dimenzí daných bodů na druhou mocninu, ale bez odmocnění. Tímto základním nastavením lze docílit větší rychlosti díky menší časové náročnosti operace, nicméně pravá Euklidovská metrika lze bez problémů nastavit. Taktéž při zvolení metody vyhledávání approximate nearest neighbor je ϵ vzdálenost počítána jako pravé Euklidovská metrika.

Nejzákladnějším objektem, se kterým se v knihovně pracuje je bod, přesněji `ANNpoint`, který je tvořen polem koordinát bez přesně udaného počtu dimenzí, jednoduše ukazatel na koordinátu:

`typedef ANNcoord* ANNpoint; // bod`

Uživatel knihovny se stará o alokaci a dealokaci úložiště bodů. Každý bod musí mít alokováno minimálně tolik komponent jaký je počet dimenzí prostoru, žádné další komponenty nejsou potřeba jsou ignorovány.

Jelikož ANN pracuje s poli bodů a vzdáleností, pro práci je potřeba alokovat také pole pro tyto dva objekty:

`typedef ANNpoint* ANNpointArray; // pole bodu`

`typedef ANNdist* ANNdistArray; // pole tzv. squared distances`

Vstupní hodnoty, které předkládáme metodám jsou tedy pole bodů a vrací se nám pole indexů, které ukazují na body v poli, které jsou nejblíže.

4.2.1 Vyhledávání nearest neighbor

Struktura, kterou knihovna využívá pro hledání NN v kd-stromu se nazývá `ANNpointSet`. Jedná se o abstraktní objekt, na kterém jsou definovány dvě operace pro vyhledávání: `annkSearch()` a `annKFRSearch()`. Tato abstraktní struktura umožňuje tři konkrétní instance `ANNbruteForce`, `ANNkd_tree` a `ANNbd_tree`. Nás pro potřeby rychlého vyhledávání pro DBSCAN algoritmus zajímá prostřední z nich `ANNkd_tree`. Instance pro hledání hrubou silou `ANNbruteForce` neslouží k efektivnímu vyhledávání nejbližšího souseda, ale jako kontrola pro pokusy s daty.

4.2.2 Konstrukce stromu

Samotná konstrukce stromu je poměrně jednoduchá. Skládá se z nahrání všech bodů do pole tvořeného `ANNpoint`, pro které si pomocí funkce alokujeme paměť:

`ANNpointArray dataPts = annAllocPts(maxPts, dim); // alokace pole pro uchovani bodu`

Následně do pole nahrajeme body a vytvoříme kd-strom:

```
ANNkd_tree* kdTree;     // struktura

kdTree = new ANNkd_tree(      // sestaveni stromu
  dataPts,           // data
  nPts,              // pocet bodu
  dim);              // dimenze prostoru
```

Výpis 3: Tvorba kd-stromu

samotná struktura kd-stromu neukládá přímo body ale ukazatele do pole `dataPts`.

Statistiky vytvořeného stromu pro 1000000 bodů s dimenzí prostoru 8 můžou vypadat například takto:

```
Build ann-structure:
  split_rule   = suggest
```

Tabulka 1: Průměrný čas konstrukce kd-stromu

Počet bodů:	2 dimenze	4 dimenze	8 dimenzí	16 dimenzí
$1*10^3$	0,001 s	0,001 s	0,001 s	0,002 s
$1*10^4$	0,012 s	0,015 s	0,023 s	0,035 s
$1*10^5$	0,18 s	0,226 s	0,325 s	0,612 s
$1*10^6$	3,355 s	4,233 s	6,251 s	11,432 s
$1*10^7$	68,07 s	84,332 s	130,513 s	n/a

Hodnoty jsou průměrem 5 konstrukcí pro každou konfiguraci. Jednotkou je sekunda. Poslední konfiguraci nelze vyzkoušet z důvodu nedostatku paměti.

```
shrink_rule  = none
data_size    = 1000000
dim          = 8
bucket_size  = 1
process_time = 5.468 sec
(Structure Statistics:
  n_nodes        = 1999999 (opt = 2000000, best if < 20000000)
     n_leaves    = 1000000 (0 contain no points)
     n_splits    = 999999
     n_shrinks   = 0
  empty_leaves   = 0 percent (best if < 50 percent)
  depth          = 25 (opt = 19, best if < 318)
  avg_aspect_ratio = 2.03102 (best if < 20)
)
```

Výpis 4: Statistiky vytvořeného kd-stromu

Jak lze vypozorovat ve výpisu statistik, sestavení stromu i pro relativně hodně bodů a při větším počtu dimenzí je velmi rychlé. Takto sestavená struktura se následně uchovává v paměti a není potřeba ji pro účely dotazů nijak měnit. Funkcionalita knihovny umožňuje uložit celou sestavenou strukturu jako soubor na pevný disk a opět ji načíst.

V tabulce 1 se můžeme podívat na porovnání časové náročnosti pro tvorbu kd-stromu v závislosti na počtu dat a dimenzí prostoru. Jak je patrné z naměřených údajů, tvorba stromu je i pro data o velikosti stovek tisíc bodů velmi rychlá a vzhledem k tomu, že nám stačí sestavit strom pouze jednou a následně ho opakovaně využívat, jedná se o velmi efektivní způsob práce s multidimenzionálními body.

4.2.3 Vyhledávání

i při velkém počtu bodů ve struktuře nepřesáhla hloubka stromu řádově nižší desítky úrovní. Dotazování se na tuto strukturu je tedy velmi rychlé. Na následující ukázce se můžeme přesvědčit,

Tabulka 2: Průměrný čas nalezení nejbližšího souseda

Velikost stromu:	2 dimenze	4 dimenze	8 dimenzí	16 dimenzí
$1*10^3$ bodů	1e-005	1e-005	3e-005	0,00018
$1*10^4$ bodů	1e-005	1e-005	7e-005	0,00189
$1*10^5$ bodů	3e-005	4e-005	0,00017	0,00568
$1*10^6$ bodů	0,00022	0,00023	0,00039	0,01634
$1*10^7$ bodů	0,00188	0,00192	0,00212	n/a

Hodnoty jsou průměrem 100 dotazů pro každou konfiguraci. Jednotkou je sekunda

že statistiky 100 dotazů na nalezení nejbližšího souseda dosahují průměrně času 0,00044 sekundy na vyhodnocení jednoho dotazu:

```
Run Queries:
  query_size   = 100
  dim          = 8
  search_method = priority
  near_neigh   = 1
  query_time   = 0.00044 sec/query
```

Výpis 5: Test dotazování

V tabulce 2 si můžeme srovnat rychlost jednotlivých dotazů v závislosti na velikosti stromu. Z údajů je patrné, že velkou roli nehraje pouze počet bodů ve stromu, ale dost podstatný parametr je počet dimenzí. Tento naměřený výsledek jednoznačně potvrzuje domněnku, že kd-strom jako datová struktura je velmi efektivní, pokud je počet dimenzí maximálně v řádu desítek. Již při velikosti prostoru 16 dimenzí je efektivita vyhledávání podstatně menší, něž při velikosti prostoru do 8 dimenzí. Tato vlastnost je způsobena podstatou algoritmu kd-stromu, tedy dělením dat v jedné dimenzi na jedné úrovni stromu. I přesto je stále možné považovat kd-strom za velmi rychlý a efektivní způsob prohledávání multidimenzionálních prostorů.

Pro účely využití kd-stromu v řešení problému shlukování nám ovšem nestačí najít pouze jednoho nejbližšího souseda k danému bodu v datasetu, ale potřebujeme obecně k nejbližších bodů. Tento počet je závislý na nastavení vstupního parametru `minPts`, který nám říká počet bodů, které musí mít *core* bod v sousedství *epsilon*. I tento problém ale kd-strom velmi efektivně řeší. Místo pouhého jednoho bodu nastavíme při dotazu parametr k, který nastavuje počet vrácených nejbližších bodů. Díky tomuto omezení tedy nikdy nebudeme zatěžovat datovou strukturu dotazy na více bodů, než kolik jich potřebujeme pro určení druhu bodu v algoritmu DBSCAN. Použití takovéto struktury a vyhledávacích možností má své výhody i nevýhody. Je velmi výhodné, že se vždy doptáváme pouze na nejmenší nutný počet sousedů, protože tímto trávíme vyhledáváním nejmenší možnou dobu. Na druhou stranu si ale musíme uvědomit, že algoritmus první najde k nejbližších sousedů a následně jejich vzdálenost porovná s hodnotou *epsilon*. Nejedná se tedy o tzv. *range searching*, tedy vyhledávání do určité vzdálenosti. Tuto vlastnost

danou použitím tohoto typu vyhledávání na struktuře je třeba mít na paměti při nastavování hodnoty k (parametr `minPts` v naší verzi algoritmu DBSCAN). Konkrétně tedy musíme nastavit větší hodnotu parametru `minPts`, chceme-li pokrýt větší oblast sousedů v rámci `epsilon` vzdálenosti. Dotaz na specifický počet bodů vypadá takto:

```
kdTree->annkSearch(
   queryPt,              // dotazovany bod
   k,                 // pocet sousedu
   nnIdx,                // pole indexu bodu (vraceno)
   dists,                   // pole vzdalenosti bodu (vraceno)
   eps);                 // mira pripustne chybovosti, defaultne 0
```

Výpis 6: Dotaz na k nejbližších bodů k bodu `queryPt`

Po zavolání této metody máme dvoje pole o velikosti `k`, přičemž jedno obsahuje ukazatele na nejbližší body a druhé pak vzdálenosti od bodu `queryPt`. Díky tomu jsme schopni dále pracovat s danými body a jejich vzdálenostmi od vyhledávaného bodu a nevyužíváme dále sestavený strom. Toto řešení nám velmi usnadňuje dále řešenou paralelizaci. Pro ukázku se můžeme podívat na vypsaný výsledek předešlého dotazu, který nalezl 5 nejbližších sousedů daného bodu v 2D prostoru:

```
Query point: (-0.532226, -0.727036)
       NN:    Index  Distance
       0      8      0.612857
       1      12     0.667689
       2      10     0.725486
       3      0      0.933152
       4      15     1.02157
```

Výpis 7: Pět nejbližších sousedů daného bodu

Na konci programu po ukončení práce se strukturou smažeme alokovaná pole a uvolníme paměť:

```
delete[] nnIdx;
delete[] dists;
delete kdTree;
annClose();   // konec ANN
```

Výpis 8: Smazání struktury a uvolnění paměti

5 DBSCAN

Shlukovací analýza je technika dolování dat, která si klade za cíl shromažďovat objekty do skupin na základě jejich podobnosti. Cílem tedy je, aby objekty v rámci jednoho shluku byly vzájemně co nejvíce podobné, a naopak co nejvíce odlišné od objektů z jiných shluků. Detailnější popis jsme si popsali v kapitole 2. Nyní se budeme věnovat algoritmu *Density-based spatial clustering of applications with noise* známému pod zkratkou *DBSCAN*.

5.1 Popis

Klasický způsob, jakým algoritmus DBSCAN funguje jsme si popsali v části 2.4. Ve své nejzákladnější podobě je algoritmus DBSCAN založen na tom, že nachází shluky objektů na základě prozkoumání jejich okolí a rozdělení na oblasti s hustým osídlením (kde se formulí shluky) a s řídkým osídlením (kde nacházíme pouze šum). Detekce šumu a výhoda nalezení předem neznámého počtu shluků jsou dva hlavní benefity tohoto algoritmu. Práce DBSCAN algoritmu spočívá v tom, že hledá hustě osídlené oblasti, ve kterých nachází *core* body, nebo také středy shluků, které tvoří jádro samotných shluků a ty poté rozšiřuje. Tímto postupným procházením bodů a rekurzivní expanzí detekovaných shluků je ovšem algoritmus závislý na sériovém zpracování a velmi těžko paralelizovatelný.

V programu je algoritmus DBSCAN řešen takto.

```
vector<int> pDBSCAN(
   const ANNpointArray& dataPts,
   ANNkd_tree* kdTree,
   const float eps,
   const size_t minPts,
   const size_t nPts);
```

Výpis 9: Hlavička metody DBSCAN

Pro běh algoritmu potřebujeme následující vstupní data a parametry:

- `dataPts` je pole obsahující body z analyzovaného datasetu. Pomocí tohoto pole se doptáváme na body v momentě, kdy s nimi potřebujeme pracovat. Je to jediné místo v programu, které po celou dobu uchovává všechny body datasetu. Ostatní proměnné a struktury obsahují buď pouze jednotlivý bod nebo odkaz na bod v tomto poli. Díky tomuto ukládáme vstupní data pouze jednou, byť samotná datová struktura, která následuje, má taktéž paměťové nároky velmi závislé na počtu bodů.

- `kdTree` tímto ukazatelem poskytujeme algoritmu odkaz na strukturu kd-stromu, který slouží pro vyhledávání nejbližších sousedů. Sestavování této struktury jsme si popsali v kapitole 4.2.2.

- `eps` jeden ze dvou vstupních parametrů pro samotný algoritmus DBSCAN. *Epsilon* je parametr vzdálenosti a určuje hranice sousedství daného bodu. Body, které leží od zkoumaného bodu do této vzdálenosti, jsou v jeho sousedství.

- `minPts` druhý z parametrů DBSCANu. Určuje kolik bodů musí ležet v sousedství bodu, aby byl klasifikován jako *core* bod. Pokud v sousedství bodu leží menší počet bodů, než kolik určuje tento parametr, může být klasifikován pouze jako *border* bod nebo *noise*. Pro nás je tento parametr nejdůležitějším parametrem algoritmu. Díky použité datové struktuře, která podporuje vyhledávání k nejbližších sousedů je to právě tento parametr, který určuje hodnotu k. Na základě jeho nastavení se tedy budeme doptávat kd-stromu na daný počet sousedů a následně jejich vzdálenost porovnáme s hodnotou parametru `eps`, abychom zjistili, zda náleží nebo nenáleží do sousedství daného bodu.

- `nPts` označuje celkový počet bodů v datasetu, respektive délku pole `dataPts`.

5.2 Paralelní programování

Abychom mohli využít paralelizaci v řešení problému algoritmu DBSCAN je potřeba jej upravit, tak ať efektivně využijeme výpočetní sílu vláken běžících současně. Mezi nejvíce používané způsoby paralelizace řadíme tři možnosti *OpenMP*, *MPI* a *CUDA*. Každý z těchto způsobů paralelizace programu se liší způsobem, jakým k paralelizaci přistupuje:

- OpenMP - *Open Multi-Processing* je API, které pracuje na většině platforem a umožňuje multiprocesní programování s použitím sdílené paměti v jazycích C, C++ a Fortran. Hlavní vlastností OpenMP je práce se sdílenou pamětí na jednom zařízení. Hodí se tedy na řešení paralelizace v rámci jednoho počítače. Lze jej chápat jako rozvětvení výpočtu v místech, kde to dává význam, přičemž jednotlivá vlákna mohou přistupovat ke stejné paměti.

- MPI - *Message Passing Interface* je API založené na zasílání zpráv mezi jednotlivými uzly, které zpracovávají kód. MPI pracuje na principu spuštění stejného programu na více procesorech. Sdílení dat se zajišťuje posíláním zpráv s daty. Funguje tedy na principu distribuované paměti. MPI není závislá na struktuře ani programovacím jazyce, nejpoužívanější je v C a Fortranu. MPI nachází své místo hlavně na výpočetně silných klastrech a superpočítačích, kde se jedná v podstatě o standard mezi technologiemi, přičemž další technologie se používají jako doplněk. Na každém procesoru běží stejný program samostatně a zpracovává svou část dat, jedná se o princip – *Single Program Multiple Data*.

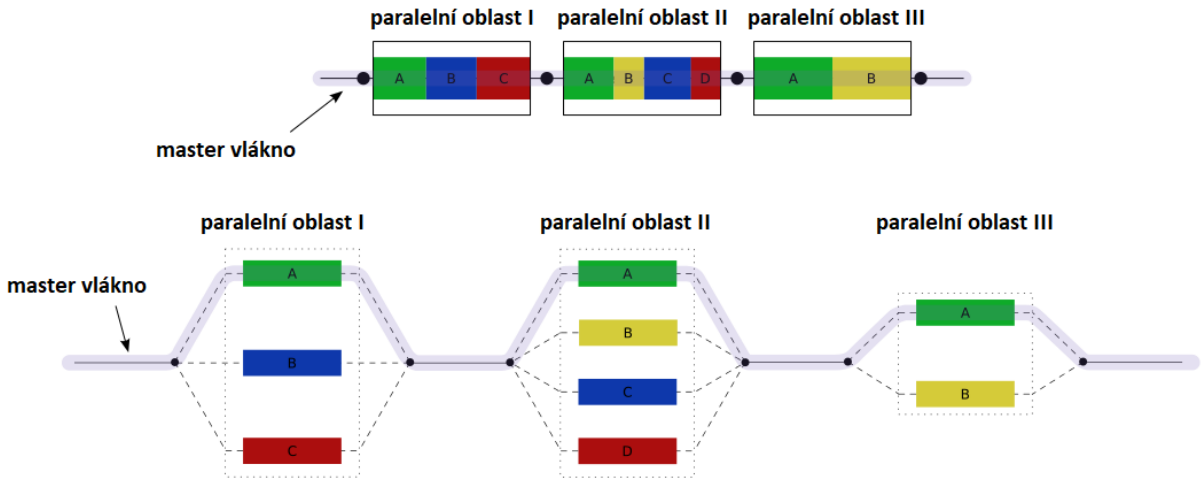
- CUDA - *Compute Unified Device Architecture* je technologie společnosti nVIDIA, která umožňuje spouštět programy v jazycích C, C++ a Fortran a programy postavené na OpenCL, DirectCompute apod. na vybraných grafických kartách společnosti nVIDIA. CUDA aplikace se skládají z částí, které běží buď na CPU nebo na grafické kartě, části běžící na grafické kartě jsou spouštěny zavoláním funkce kernel, kterou provádí každé vlákno.

Paměť pro paralelní zpracování je odvozena od možností grafických karet a umožňuje sdílet data, která se nahrávají do zařízení a po ukončení výpočtů zpět do operační paměti počítače. Z hlediska práce s pamětí zde existuje několik typů druhů paměti, které mohou být sdíleny napříč celým programem, sdílení uvnitř bloků vláken, paměť pouze pro dané vlákno a další. Vlákna jsou na grafické kartě organizována do 1D, 2D nebo 3D bloků, vlákna v rámci bloků mohou sdílet data a synchronizovat svůj běh. Bloky jsou organizovány do 1D, 2D nebo 3D mřížky a pracují nezávisle na sobě.

Při porovnávání technologií popsaných výše jsme brali v potaz problém, který budeme paralelizací DBSCANu řešit a možnosti, které se nám nabízejí. Jako nejlepší možností pro naši situaci se jeví použití technologie OpenMP. Protože výsledná paralelizace poběží v rámci jednoho počítače je žádoucí využít jeden z principů pracujících se sdílenou pamětí. Paměť obsahující body z datasetu a datové struktury vytvořené kd-stromem budou sdíleny a fyzicky budou v paměti počítače, přičemž jednotlivá vlákna k nim budou přistupovat podle potřeby. Základ algoritmu DBSCAN bude běžet sériově a některé jeho výpočetní části paralelizujeme. Tento přístup by měl zajistit rychlejší běh algoritmu v rámci jednoho procesoru a zároveň zachovat relativní jednoduchost paralelizace v jazyce C++.

5.2.1 OpenMP

Podívejme se nyní na možnosti, které nám technologie OpenMP nabízí pro paralelní programování a následně se pokusíme tyto možnosti využít při řešení našeho problému.



Obrázek 16: Ilustrace principu, na kterém funguje model fork-join, který využívá OpenMP[6]

[6]Obrázek převzat z: OpenMP. *Wikipedia* [online]. [cit. 22.4.2018]. Dostupné z: `https://en.wikipedia.org/wiki/OpenMP`

Hlavní vlákno (master thread) provádí klasický sekvenční průchod programem a podle potřeby vytváří skupinu podvláken, která zpracovávají části paralelizovaných sekcí programu. Tento princip je názorně ukázán na obrázku 16, kde máme vyznačeno hlavní vlákno *master*, které při průchodu paralelní sekcí vytvoří další vlákna a na konci sekce je opět ukončí.

Jak jsme si popsali v předchozí kapitole 5.2 vykonávání programu je v OpenMP rozděleno mezi paralelizovanou část a část vykonávanou klasicky jedním vláknem. V rámci OpenMP lze používat následující příkazy:

- Parallel - blok kódu uvnitř tohoto příkazu je prováděn několika vlákny

- Loop - tyto iterace, typicky smyčka `for` jsou prováděny paralelně několika vlákny. Podle režimu plánování může být smyčka rozdělena do několika částí kdy každé vlákno vykonává jednu z částí, případně lze použít dynamický režim, kdy vlákno dostane smyčku k provedení a po jejím dokončení je mu přiřazena další nevykonaná smyčka. Druhý z přístupů má větší režijní náklady, ale je vhodný v částech kódu, kde se výpočetní a časová náročnost mezi smyčkami může velmi lišit. Na konci provádění smyčky je implicitní bariéra, kdy na sebe vlákna čekají. Toto chování lze změnit příkazem `nowait`. Velmi dobře paralelizovatelné jsou smyčky, kdy je předem znám počet iterací a nemění se v průběhu běhu programu, jedná se o smyčku `for`. Dalším z předpokladů je nezávislost iterace na žádné další a nezávislost dat. Naopak podmíněné smyčky, například smyčka typu `while` není pro paralelizaci vhodná. Další těžko paralelizovatelné konstrukce jsou ty, kde jsou iterace navzájem závislé ať už jedna k druhé, nebo přes závislá data. Sekci uvozujeme příkazem `omp parallel for`.

- Sections - pomocí tohoto příkazu lze paralelizovat program jiným způsobem (neiterativně) na rozdíl od příkazu smyčkových. Blok kódu rozdělíme pomocí příkazu `omp section` do částí, kde každou sekci vykoná jedno vlákno. Nebude se tedy vykonávat stejný kód, ale program se rozdělí a jednotlivé části se vykonají současně.

- Task - tento příkaz rozšiřuje možnosti paralelizace některých hůře paralelizovatelných částí. Když vlákno zaznamená konstrukci task, vygeneruje se nový task a systém rozhodne, zda se vykoná ihned nebo jej odloží na později. Dokončení tasku může být vynuceno synchronizací. Blok vytváříme pomocí příkazu `omp task` a jeho synchronizaci pak pomocí `omp barrier` nebo `omp taskwait`. Konstrukce obalená blokem task se spustí na pozadí a runtime system rozhodne, kdy přesně bude vykonána. Tento způsob se hodí pro vykonávání nezávislého kódu, u kterého nám nejde o přesnou posloupnost vykonání.

- Synchronization - synchronizaci používáme v kritických sekcích programu, tam kde nám záleží na závislostech (některá část musí být hotova dřív než jiná) nebo v případě, že některé sekce nemohou být vykonány paralelně. Způsobů synchronizace je několik, řadíme do nich příkazy `Critical`, `Atomic`, `Barrier`, `Single`, `Ordered` a `Flush`.

  - `Critical` definuje sekci, kterou může vykonávat současně pouze jedno vlákno, přičemž je jedno které a všechna vlákna postupně sekcí projdou. Uvozujeme ji příkazem `omp critical`.
  - `Atomic` specifikuje část programu, kterou může vykonávat nejvýše jedno vlákno programu, opět je jedno které. Tato část bude vykonána pouze jednou. Příkaz pro použití je `omp atomic`.
  - `Single`, `Master` fungují podobně jako předchozí příkazy s tím rozdílem, že u příkazu `omp master` bude blok vykonán právě jedním vláknem, a to master vláknem. `omp single` specifikuje sekci, která se provede pouze jedním vláknem, a to pouze jednou.
  - `Barrier` funguje jako klasická bariéra, která synchronizuje všechna vlákna. Uvozuje se příkazem `omp barrier`.
  - `Ordered` zajišťuje, že blok bude vykonán ve stejném pořadí, jako by byl vykonán v serializovaném běhu programu. Uvozuje se příkazem `omp ordered`.
  - `Flush` zajišťuje konzistentní pohled na objekty v paměti, funguje tedy jako paměťová bariéra. Začíná příkazem `omp flush`.

- Reduction - poslední z příkazů redukuje seznam proměnných z jednotlivých vláken do jedné, a to za použití stanovených operátorů. Lze jej použít pro operátory +, -, *, &, ^, |, && a ||. Je vhodný, pokud řešíme paralelizací problém sčítání, násobení apod. a chceme z paralelní části dostat danou proměnnou. Příkaz je `omp reduction (op : list)`.

Pokud v programu využíváme OpenMP, musíme vložit do programu include na jeho hlavičkový soubor `#include <omp.h>` a dát vědět překladači, že jej budeme používat. Ve Visual Studiu je tato možnost v nastavení *Project > Properties > C/C++ > Language > Open MP Support* nastavíme na `Yes (/openmp)`.

OpenMP se spouští pomocí direktiv, proto před každou instrukcí musíme použít `#pragma`, jako příklad si uvedeme nejjednodušší paralelní blok OpenMP, který nám vypíše text *Hello World from thread:* a číslo vlákna:

```
int tid; // tato promenna bude privatni pro kazde vlakno
#pragma omp parallel private(tid)
{
   tid = omp_get_thread_num(); // zjistime cislo vlakna
   printf("Hello World from thread: %d\n", tid); // vypiseme
}
```

Výpis 10: Paralelní blok hello world

Podle počtu vláken následně na výpisu nalezneme například:

```
Hello World from thread: 0
```

```
Hello World from thread: 2
Hello World from thread: 1
Hello World from thread: 3
```

Výpis 11: Hello World od každého vlákna

Je patrné, že tento kód běžel na čtyřech vláknech. Všimněme si, že vlákna se nevykonala postupně za sebou dle svého pořadí, ale vlákno 2 bylo vykonáno již před vláknem 1.

5.3 Paralelizace DBSCAN algoritmu

Vzhledem k faktu, že je ve své klasické podobě algoritmus DBSCAN závislý na sériovém zpracování bodů v datasetu, budeme muset algoritmus upravit tak, ať jeho paralelizace dává význam z hlediska rychlejšího běhu programu. Zároveň by měl takový způsob paralelizace využít možnosti OpenMP paralelních bloků a rovnoměrně zatížit jednotlivá vlákna.

Detailně jsme se teoretickému popisu algoritmu věnovali v sekci 2.4, nyní si tedy jen abstraktně připomeňme kroky algoritmu:

1. Navštívení prvního bodu z datasetu (nebo bodu, který jsme doposud nenavštívili) a vyhodnotit, zda se jedná o *core* bod, *border* bod nebo *noise* na základě vstupního parametru `minPts` a `eps`.

2. U *core point* bodů rekurzivně aplikovat stejné pravidlo a expandovat shluk. Ostatní typy bodů ignorovat.

3. Přiřadit každý *border* bod blízkému shluku, pokud se nachází vzdálenosti *epsilon*, jinak označit bod jako *noise*.

Problémem u klasické sekvenční verze algoritmu je provázanost těchto tří kroků, kdy vždy hledáme *core* bod a zanořujeme se do něj ihned po jeho detekci a postupným hledáním komponenty v grafu expandujeme shluk. Pokud již žádný další způsob rozšíření shluku nenajdeme, přesuneme se k dalšímu bodu v datasetu a opakujeme postup. Shluky tedy hledáme postupně jeden po druhém a jakmile jej detekujeme, rozšíříme je na maximální úroveň.

Abychom byli schopni využít možnosti paralelizace programu, které nabízí OpenMP, potřebujeme ideálně rozdělit sekvenční průběh algoritmu na několik částí. V těchto částech by mělo docházet k výpočetním operacím, které urychlíme současným zpracováním několika vláken najednou. Důležité také je, aby v těchto částech probíhal zápis do paměti vždy jen pro daný zkoumaný bod, který bude u každého vlákna odlišný a vyhnout se tak problémům se souběhem.

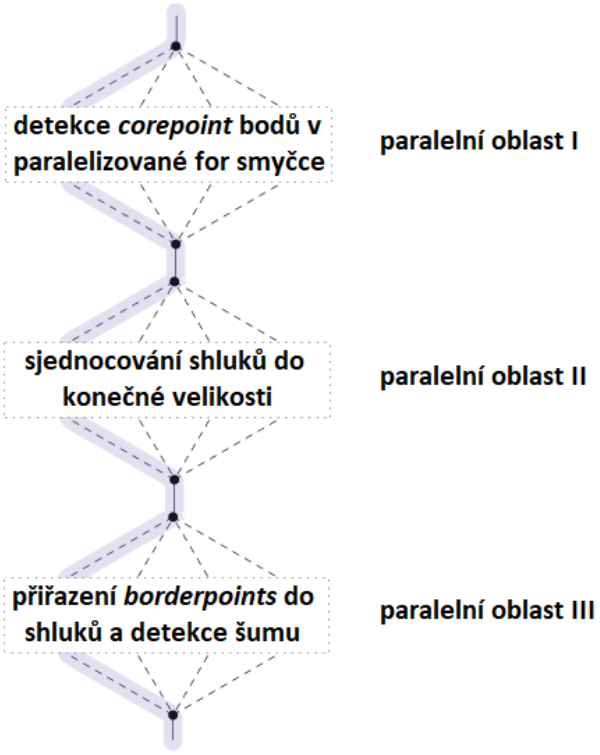
Pokusíme se upravit algoritmus tak, že jednotlivé kroky rozdělíme do samostatných částí, které se nebudou navzájem provazovat. První vždy dokončíme daný krok pro všechny body z datasetu a jakmile se tak stane, začneme se zpracováním další částí. Tímto nám vzniknou opět tři kroky, které ale nebudou vzájemně provázané, jejich zpracování bude vždy závislé na dokončení předešlého kroku:

1. Projdeme všechny body v datasetu, dotážeme se na jejich sousedy a vzdálenost k nim porovnáme s `eps` vzdáleností. Pokud nalezneme *core* bod zapíšeme si tuto informaci do pole na index daného bodu. Zároveň s tímto vytvoříme nový shluk, který bude nést ID totožné s indexem bodu v poli.

2. Všechny body typu *core* projdeme znovu stejným způsobem, zaznamenáme si `core` body v jejich sousedství a poznamenáme si všechna ID jejich shluků. Tato ID setřídíme od nejmenšího po největší a všem *core* bodům ze sousedství i našemu zkoumanému nastavíme jako ID shluku nejmenší hodnotu z celého seznamu. Zároveň si v každé jednotlivé smyčce značíme, zda došlo k přepsání některého ID shluku (sloučení s jiným shlukem) a pokračujeme až dokud nedojde v celém datasetu k žádnému sloučení.

3. Projdeme všechny body, které nejsou typu *core* a zjistíme, zda je v jejich sousedství nějaký *core* bod. Pokud ano, označíme bod jako *border* a přiřadíme jej do daného shluku. Jinak jej označíme jako *noise*.

Pokud se podíváme na to, co jednotlivé body dělají a jakým způsobem zapisují do paměti, zjistíme, že všechny tři jdou paralelizovat alespoň částečně ve formě cyklů. U bodu 1 a 3 je situace poměrně snadná, v jednom vláknu zkoumáme vždy jeden bod a zjištěný poznatek zapisujeme na jeho místo v poli. Nezasahujeme do jiných částí pole a nepřepisujeme navzájem žádnou hodnotu. Rozdělení algoritmu na tři paralelní částí je znázorněno na obrázku 17.



Obrázek 17: Rozdělení algoritmu DBSCAN na tři paralelizované části

Nejsložitější částí je druhá sekce. V této části programu máme z předchozí části detekované body typu *core*, přičemž každý z nich také založil nový shluk s ID, které je rovno jeho pořadí v poli. V druhé paralelně běžící části pak musíme sjednotit do jednoho shluku ty body, které jsou vzájemně dosažitelné mezi sebou. Jak by mohla vypadat procedura pro slučování shluků je naznačeno níže:

```
void mergeClusters(data, point, minPts, eps):
   nb = getNeighbours(point, minPts, eps)
   corePtsIdxs.add(point.index)
   clusters.add(point.clusterID)
   for each p in nb
      if isCorepoint(p)
         corePtsIdx.add(p.index)
         clusters.add(p.clusterID)

   if length(clusters) > 1
      sort(clusters)
      for each idx in corePtsIdxs
         data[idx].clusterID = clusters.first
```

Výpis 12: Princip sjednocování shluků zapsaný v pseudokódu

Tímto způsobem se postupně formují jednotlivé shluky spojováním menších částí. Každé vlákno se věnuje jednomu bodu, čte data ostatních bodů a následně zapisuje potenciálně několika bodům nové ID shluku. V praxi to ovšem nezpůsobuje problém, protože tyto body postupně konvergují do jednoho shluku. Jejich dočasné přeřazení (pokud k němu vůbec dojde) mezi podmnožinami tohoto finálního shluku nečiní problém, protože nakonec budou všechny součástí společného celku. Algoritmus tímto způsobem postupně projíždí všechny body typu *core point* a zapisuje si, zda došlo v celém cyklu k sjednocení nějakých shluků, to vše se děje uvnitř velkého `do{}while()` cyklu, kde právě podmínku tvoří informace o tom, zda došlo ke sloučení shluků či nikoliv. Pokud nedošlo v žádným změnám tak cyklus končí.

Následující výpis z kódu obsahuje vnější cyklus do-while, který obaluje paralelní oblast II. Flag `changed` slouží k přenosu informace o tom, zda došlo při průchodu datasetem ke sloučení shluků. Z důvodu velmi dlouhého kódu, který není podstatný pro demonstraci tohoto principu byly některé části vyňaty a jejich činnost je zapsaná pouze abstraktně v komentáři:

```
bool changed = false;
do{
   changed = false;
   #pragma omp parallel for schedule(dynamic)
   for (int i = 0; i < numPoints; ++i) // prechazeni bodu
   {
```

```
      if (pointTypeCluster[i] != 1){ // prochazime pouze core point body
         continue;
      }

      // ...
      // hledani vsech shluku v okoli a trizeni ID nalezenych shluku
      // pokud doslo ke slouceni shluku, nastavime zaroven flag:

      changed = true; // doslo ke slouceni shluku
   }
} while (changed);
```

Výpis 13: Vnější do-while cyklus obalující paralelní oblast II

U všech paralelních částí algoritmu používáme dynamický systém plánování nastavený v direktivě OpenMP:

`#pragma omp parallel for schedule(dynamic)`

Tento systém plánovaní způsobuje, že každé vlákno dostane další práci po ukončení svého současného úkolu a nečeká se na synchronizaci všech vláken po provedení iterace. Režim `dynamic` má větší režijní náklady než režim `static` a používá se proto v případech, kdy se doba vykonávání iterací mezi sebou podstatně liší. V našem případě vede použití tohoto režimu plánování k empiricky zjištěnému zrychlení běhu o zhruba 30 %.

Proměnné deklarované jako `const` ať už v parametrech funkce nebo v těle funkce před začátkem paralelních bloků jsou automaticky označeny jako sdílené. Jelikož nemohou být modifikovány je logické, že každé jádro nebude mít svou kopii a bude číst z jedné proměnné. Příkladem takovýchto proměnných je například `eps` a `minPts`, které jsou vstupními parametry pro DBSCAN. Dále `nPts` označující počet bodů a `space` což je pole obsahující body a jejich souřadnice. Všechny tyto proměnné a pole slouží pouze pro čtení a nejsou přepisovány. Není tedy efektivní ani žádoucí, aby si každé vlákno dělalo svou kopii.

Proměnné deklarované v rámci paralelních oblastí jsou privátní a každé vlákno má svou vlastní proměnnou dostupnou pouze v rámci vlákna. Příkladem těchto proměnných jsou kopie bodu, který si dané vlákno zkopíruje ze sdíleného pole `space` a se kterým potom pracuje v rámci bloku pouze dané vlákno. Dále to jsou pomocné proměnné a vektory sloužící pro ukládání indexů sousedních bodů, ID jednotlivých shluků v sousedství a další pomocné proměnné deklarované uvnitř paralelní oblasti.

V rámci paralelní sekce je občas potřeba zajistit, že k dané části kódu bude v jeden čas přistupovat pouze jedno vlákno. K tomuto slouží direktiva `critical`. Tuto sekci zpracuje každé z vláken, ale pouze jedno vlákno ji bude zpracovávat v jednom okamžiku. Příkladem takové sekce je například:

```
#pragma omp critical
```

```
kdTree->annkSearch(
   queryPt, // bod
   k,       // pocet nejblizsich sousedu
   nnIdx,  // pole indexu nejblizsich sousedu
   dists,     // pole vzdalenosti k sousedum
   errorTol); // mira povolene chybovosti, defaultne 0
```

Výpis 14: Kritická sekce v OpenMP

6 Naměřené hodnoty

6.1 Testovací data

Následující měření budeme provádět na různých datech, tak abychom dosáhli co možná nejlepšího porovnání a zvýraznění odlišností v závislosti na počtu dat, dimenzí prostoru a také jejich rozložení. Pro testování naplnění datasetu a porovnání běhů algoritmů s dotazováním na nejbližší sousedy pomocí hrubé síly a kd-stromu jsme nejprve použili data náhodně vygenerovaná s rozložením každé ze souřadnic pomocí Gaussovy (normální) distribuce. Pro potřeby porovnání různého počtu dimenzí jsme si vygenerovali vždy potřebná data. Příklad, jak vypadá jeden záznam v datasetu o 8 dimenzích:

```
0.919735 -0.750654 0.176572 -0.802977 -0.0707223 -0.320515 -0.110584 0.473198
```

Následně jsme použili pro některá porovnání data z datasetu SUSY nebo jejich část. Celý dataset obsahuje několik miliónů bodů v až 19 dimenzích. Tento dataset, respektive jeho část, je výsledkem měření na částicovém detektoru v urychlovači (prvních 8 dimenzí). S tím, že dalších 11 dimenzí jsou funkcí prvních 8 (poslední z nich je pak výstupem klasifikace signál/šum). Pro představu co znamenají jednotlivé sloupce v datasetu se můžeme podívat na jejich názvy: *lepton 1 pT, lepton 1 eta, lepton 1 phi, lepton 2 pT, lepton 2 eta, lepton 2 phi, missing energy magnitude, missing energy phi, MET_rel, axial MET, M_R, M_TR_2, R, MT2, S_R, M_Delta_R, dPhi_r_b, cos(theta_r1)*. Výňatek z datasetu obsahující prvních 9 dimenzí, mezi kterými jsou výsledky měření vypadá takto (jeden záznam):

```
5.782864689826965332e-01, -6.896522045135498047e-01, -3.900941908359527588e-01,
4.800612926483154297e-01, -6.322193741798400879e-01, 1.212004899978637695e+00,
6.404729485511779785e-01, -1.622401356697082520e+00, 0.000000000000000000e+00
```

Hodnoty jsou zapsány vědeckým způsobem zapsání čísel, nicméně pro načtení do našeho programu to nehraje roli. SUSY dataset obsahuje celkově 5 milionů bodů v 19 dimenzích, v našich měřeních budeme často pracovat s výsekem z těchto bodů ať už se jedná o počet bodů nebo počet dimenzí. Na stránkách *UCi Machine Learning Repository* se můžeme podívat na kompletní informace o datech včetně publikací [7].

6.2 Porovnání hrubá síla vs. kd-strom

Podívejme se na efekt, který přineslo využití kd-stromu. Hledání nejbližšího souseda v datasetu o velikosti několika tisíců či desetitisíců bodů je pro běžný přistup hrubou silou časově náročný problém. Se vzrůstajícími daty vzrůstá také časová náročnost algoritmu. Pro vyhledávání k nejbližších sousedů potřebujeme nejprve projít celý dataset, pro každý z objektů vypočítat vzdálenost s naším bodem a toto celé aplikovat pro každý z k bodů. Dostaneme se tedy na náročnost $O(nd + kn)$, kde nd je výpočet vzdálenosti pro každý z bodů v datasetu na to celé pro všechny vyhledávané body kn. V tabulce 3 si můžeme všimnout rozdílu v nárůstu času potřebného pro běh algoritmu s použitím metody hledání k sousedů pomocí hrubé síly a kd-

Tabulka 3: Srovnání délky běhu alg. DBSCAN s využitím hrubé síly a kd-stromu.

Počet bodů:	2D hrubá síla	2D kd-strom	8D hrubá síla	8D kd-strom
100	0,013	0,014	0,003	0,003
500	0,073	0,065	0,018	0,019
1000	0,193	0,126	0,061	0,054
5000	2,033	0,653	1,214	0,621
10000	6,576	1,301	5,693	1,894
20000	18,706	2,665	28,116	6,095
50000	153,284	6,483	233,621	6,737

Hodnoty jsou průměrem 5 běhů algoritmu pro každou konfiguraci. Jednotkou je sekunda.

stromu. Časová složitost vyhledávání v kd-stromu narůstá díky předzpracování struktury daleko pomaleji, vyhledávání v kd-stromu má složitost rovnu $O(k \log n)$ [8].

Podíváme-li se na tabulku a následně na graf 18, je tento rozdíl v časové náročnosti velmi znatelný. Jako data pro běh jsme použili vygenerované body v dvou a osmi dimenzionálních prostorech, vygenerované pomocí Gaussovy distribuce, následně pak také data ze SUSY datasetu popsaného výše. Oba přístupy k hledání měli shodná data a nastavení programu. Rozdíly v čase běhu jsou tedy dány pouze přístupem k vyhledávání k nejbližších sousedů.

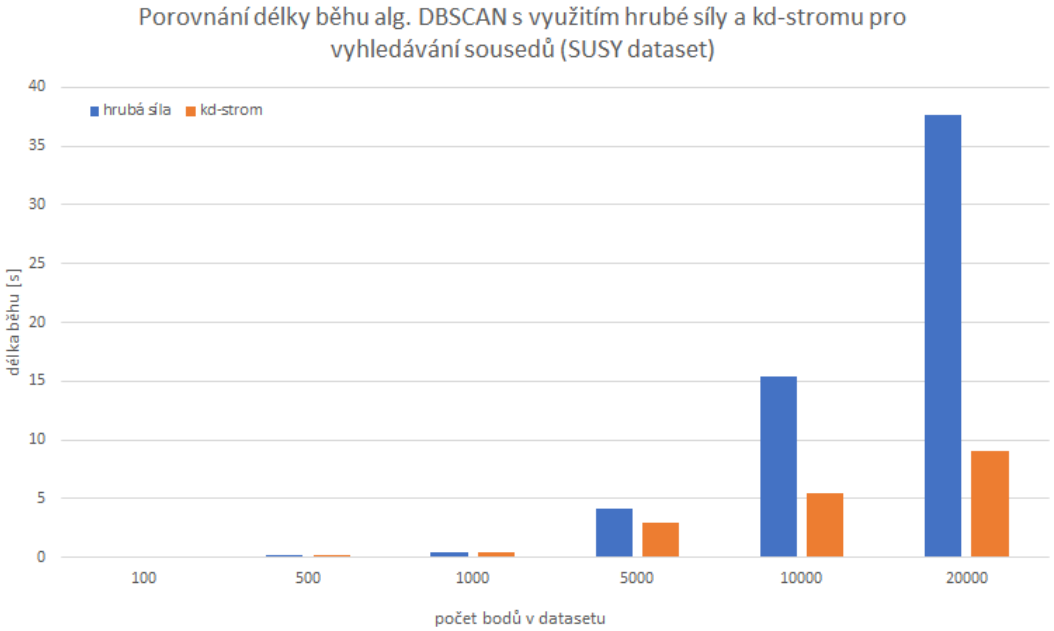
Z tabulky i obrázků vyplívá, že již pro velikost datasetu v řádu jednotek tisíc bodů, je metoda hrubé síly výrazně pomalejší. Se vzrůstajícím počtem dat je, podle složitosti uvedené výše, metoda hrubou silou výrazně pomalejší. Jako jediné východisko pro řešení problémů shlukování algoritmem DBSCAN pro velká data je použití pokročilé a předzpracované datové struktury, jejíž náročnost pro vyhledávání je nižší. Graficky jsou data z tabulky zobrazena na grafech v obrázků, poslední úsek měření pro 50000 bodů byl v grafech záměrně vynechán, neboť by svou velikostí činil ostatní měření špatně čitelná.

Z dat také vidíme, jakým způsobem ovlivňuje jednotlivé metody počet dimenzí prostoru. Porovnání 2D a 8D je u metody hrubé síly i kd-stromu podobný, byť s menším rozdílem ve prospěch druhého přístupu. Počet dimenzí tvoří větší rozdíl pro zpracování hrubou silou, což zhruba odpovídá nárůstu časové složitosti dle vzorce uvedeného výše. U struktury kd-stromu je rozdíl znatelný, ale ne tak podstatný. Jak jsme si ukázali v kapitole 4.2.3 rychlost dotazování kd-stromu se podstatně snižuje se vzrůstajícím počtem dimenzí, počet dimenzí v těchto měřeních ale ještě není tak veliký, aby se tato negativní vlastnosti kd-stromu výrazněji potvrdila. Data naměřená na vygenerovaném datasetu se nám následně potvrdila také při testování na reálném SUSY datasetu, kde si v tabulce můžeme všimnou shodných výsledků.

Tabulka 4: Porovnání hrubé síly a kd-stromu na části SUSY datasetu.

Počet bodů:	Hrubá síla	kd-strom
100	0,04	0,1
500	0,19	0,27
1000	0,47	0,43
5000	4,12	2,96
10000	15,37	5,5
20000	37,68	9,02

Hodnoty jsou průměrem 5 běhů algoritmu pro každou konfiguraci. Jednotkou je sekunda.



Obrázek 18: Porovnání hrubé síly a kd-stromu.

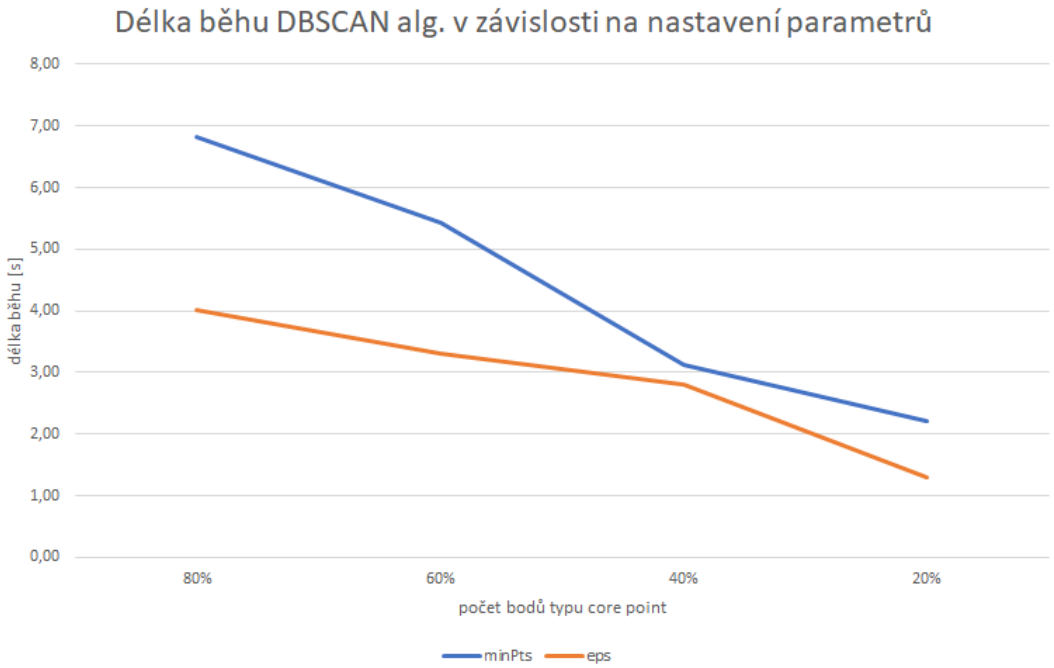
6.3 Paralelní vs. sekvenční DBSCAN

V této části si porovnáme efekt paralelizace pomocí OpenMP na naši verzi DBSCAN algoritmu. Podíváme se nejen na samotný efekt paralelizace, ale také na to, jakým způsobem běh algoritmu ovlivňují jednotlivé parametry a jak parametry ovlivňují samotnou paralelizaci.

6.3.1 Vliv parametrů na běh algoritmu

Samotnou práci algoritmu DBSCAN v jeho klasické podobě tak, jak jsme si ho popsali v části 2.4, i v naší upravené verzi velmi ovlivňuje nastavení jeho dvou parametrů `eps` a `minPts`. Tato vlastnost je pro algoritmus DBSCAN specifická, neboť špatné nastavení jednoho z parametrů bude mít za následek například nulový počet *core points* a tím pádem se nevytvoří žádný shluk

a algoritmus velmi rychle dokončí svou práci. V naší upravené verze, tak jak je zobrazena a vysvětlena v předchozích sekcích je projev těchto vlastností ještě větší. Jak si můžeme všimnout na obrázku 19 je viditelná závislost délky běhu algoritmu DBSCAN na nastavení parametrů. Na ose x vidíme vždy kolik procent datasetu jsme klasifikovali jako *core* bod. Na grafu jasně vidíme, že čím více *core* bodů máme, tím je délka běhu delší. Je to způsobeno úpravou algoritmu, konkrétně jeho druhým krokem. V tomto kroku dochází ke slévání jednotlivých shluků, které každý *core* bod vytvořil v prvním kroku. Celý dataset je opakovaně procházen až dokud již nedojde ke sloučení žádných shluků. V závislosti na nastavení parametrů pak samozřejmě ovlivňujeme počet *core* bodů a tím se tato druhá fáze prodlužuje či zkracuje. Pokud se budeme snažit dodržet počet *core* bodů přibližně na úrovní 80 %, tak je rozložení jednotlivých kroků z hlediska délky běhu přehledně zobrazeno v tabulce 5 na grafu 20.



Obrázek 19: Vliv nastavení parametrů na délku běhu algoritmu DBSCAN

Z těchto naměřených výsledků jasně vyplívá, že nejdelší část zabere prostřední krok, jak jsme vysvětlili výše. Podíváme-li se opět na graf 19 o vlivu parametrů na délku běhu, jasně nám vyplyne závěr o vlivu parametrů na délku běhu. Druhý krok algoritmu, který má za cíl dát dohromady jednotlivé shluky pomocí dříve detekovaných *core* bodů, tvoří při ideálním nastavení (80 % *core* bodů) drtivou část délky běhu algoritmu.

Toto zjištění je také jednou ze slabin naší úpravy algoritmu DBSCAN. První krok, tedy detekce *core* bodů je velmi rychlá, podobně i poslední krok, kdy již pouze přiřazujeme *border* body do shluků a označujeme zbylé body jako *noise*. Obě tyto fáze fungují tím způsobem, že projdou dataset pouze jednou a každý bod buď zpracují, nebo nechají být. Jejich vykonání tedy

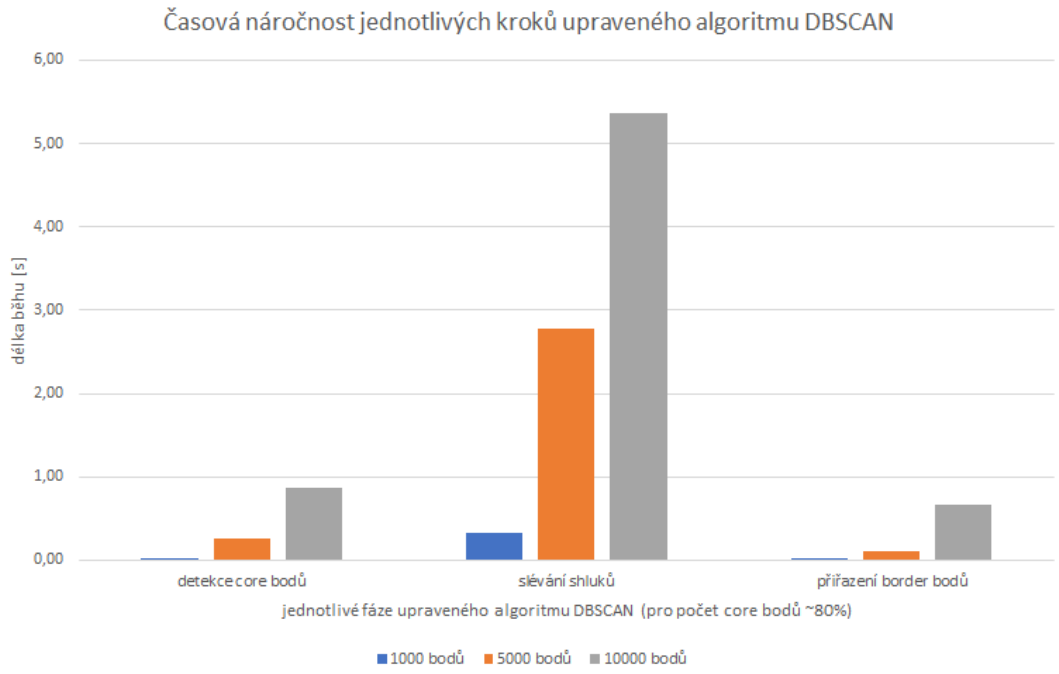
Tabulka 5: Porovnání časové náročnosti jednotlivých kroků upraveného algoritmu DBSCAN.

počet bodů	detekce core bodů	slévání shluků	přiřazení border bodů
1000	0,02	0,33	0.01
5000	0.26	2,78	0.11
10000	0.87	5,36	0,66

Hodnoty jsou průměrem 5 běhů algoritmu pro každou konfiguraci. Jednotkou je sekunda.

tvoří zanedbatelný čas v porovnání s částí prostřední, která neustále prochází všechny body (tedy všechny body typu *core*) za účelem slévání podmnožin shluků dohromady.

Rozdělení algoritmu na tyto tři části jsme provedli z důvodu snadnější paralelizace celého algoritmu, který je ve své klasické podobě závislý na jednotlivém procházení bodů v datasetu sekvenčním způsobem. Toto rozdělení na tři části se ukázalo jako relativně vhodné pro paralelizaci (každý z těchto kroků lze paralelizovat tak, ať jedno vlákno zpracovává jeden bod, případně jeho okolí), ovšem za cenu většího počtu navštívení každého bodu (a to právě v druhém kroku algoritmu).

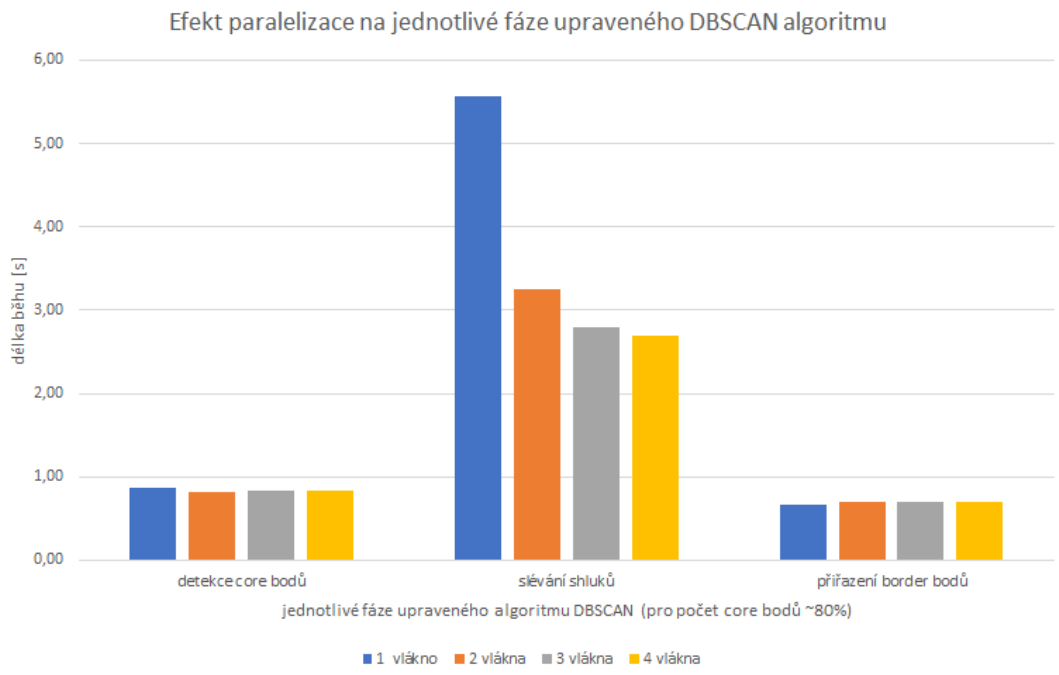


Obrázek 20: Porovnání časové náročnosti jednotlivých kroků upraveného algoritmu DBSCAN.

6.3.2 Výsledky paralelizace

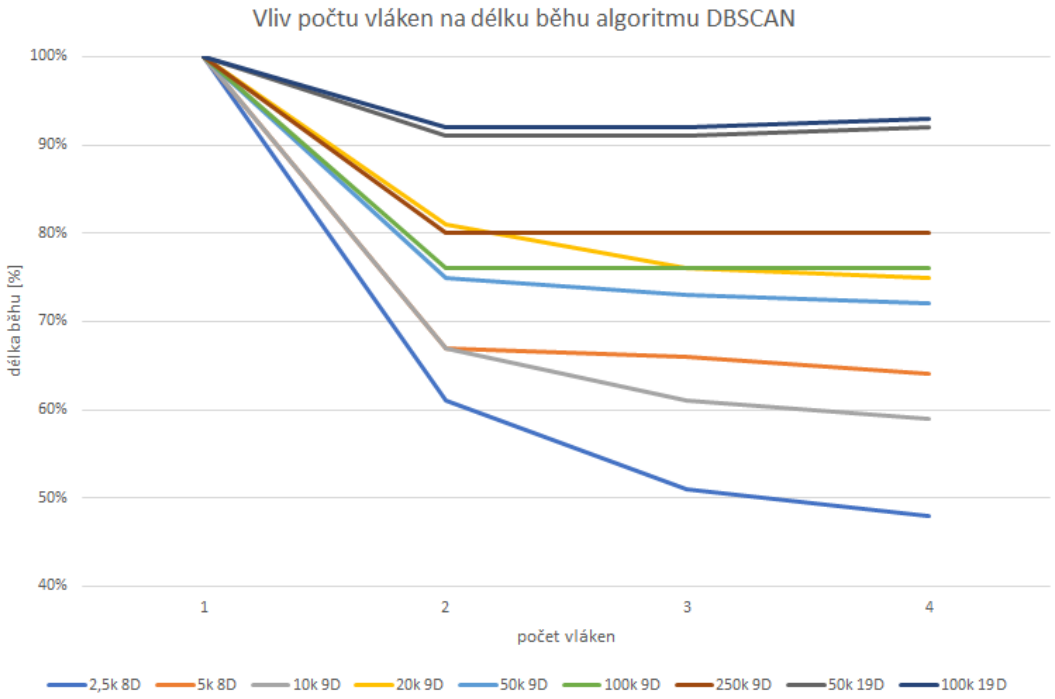
Podívejme se jakým způsobem se do běhu algoritmu promítne jeho spuštění na více vláknech. Řešení detailněji popsané v části funguje na principu tří smyček `for` z nichž každá je paralelizovaná pomocí OpenMP deklarace `#pragma omp parallel for`.

Na samotnou účinnost paralelního běhu bude mít vliv hlavně podíl části, která bude vykonávána paralelně, ve srovnání s neparalelizovanou a pak také režijní náklady samotné paralelizace. Jak již víme, přístup ke struktuře kd-tree jsme museli obalit do direktivy `#pragma omp critical` a tuto část tedy může vykonávat pouze jedno vlákno v jeden čas. Jak si můžeme všimnout na grafu 21, potvrzuje se nám zde předpoklad, že první a třetí krok algoritmu pomocí paralelního zpracování velmi mnoho času neušetří. Operace prováděné v těchto krocích spočívají v dotazu na okolní body, tento krok je ale pouze sériově zpracováván z důvodu obalení v sekci `critical`. Ve třetím kroku lze pozorovat dokonce drobný nárůst času, který je pravděpodobně způsoben režijními náklady na vytvoření a práci s několika vlákny. Největší efekt má podle předpokladů paralelizace druhé části algoritmu, tedy slévání jednotlivých shluků. Zde totiž kromě neparalelizovatelné části (dotaz do kd-stromu na okolí) také počítáme shluky a kontrolujeme který z bodů v našem okolí již v nějakém shluku je a následně pomocí primitivní logiky ze zjištěných možností vybereme vhodné ID shluku (shluk s nejmenším ID ze všech bodů v okolí, včetně našeho). Tato logika je tedy paralelizací urychlena. Z hlediska efektu paralelizace a počtu vláken je pak ale opět patrné, že hlavní část, tedy dotaz do kd-stromu, je vykonáván sériově. Jelikož se jedná o jednu z nejdéle trvajících operací v daném kroku, jeho nemožnost paralelizace celou operaci ovlivňuje. U dvou vláken dochází k výraznému nárůstu rychlosti a úspoře času. S dalšími vlákny už ale toto zrychlení výrazně zpomalilo a u 4 vláken se nám opět projevují režijní náklady, které zhruba vyrovnají úsporu výpočtu pomocí paralelizace.



Obrázek 21: Efekt paralelizace na délku běhu jednotlivých kroků algoritmu (SUSY, 10k b, minPts = 5, eps = 0.5)

Konečně v tabulce 6 (součástí přílohy) si můžeme porovnat finální efekt paralelizace. Data jsou v grafu 22 vyjádřena percentuálně a přehledně zobrazují rozdíly na celkové délce běhu algoritmu DBSCAN na různých datasetech. Všechna tato měření proběhla s nastavením algoritmu tak, ať přibližně 80 % bodů z datasetu klasifikujeme jako *core* body. Pokud bychom snížili počet těchto bodů, bude algoritmus běžet samozřejmě rychleji, získáme ale pravděpodobně horší výsledky shlukování. Na grafu lze vidět, že ohledně počtu vláken, je situace velmi podobná na všech datasetech. Mírné odlišnosti si můžeme všimnout směrem k narůstajícímu počtu dat a hlavně dimenzí, kdy se vzrůstajícím počtem vláken (3 a více) již zlepšení prakticky není detekovatelné. Tento efekt je způsoben delším trváním dotazu do kd-stromu, který samozřejmě při více bodech trvá delší dobu (stejně jako při větším počtu dimenzí), zatímco délka ostatních operací zůstává v podstatě totožná. Neparalelizovatelná část se tedy percentuálně zvětšuje (čas potřebný k jejímu vykonání) zatímco paralelizovatelná se snižuje. Do tohoto stavu pak vstupuje opět režie obhospodařující více vláken a efekt rychlejšího výpočtu postupně dorovnává.



Obrázek 22: Vliv počtu vláken na délku běhu algoritmu v závislosti na různých datech (v legendě je uveden počet bodů a dimezí).

7 Možnosti rozšíření a jiná řešení

Stávající řešení je funkční a z naměřených výsledků vyplívá, že díky paralelizaci jsme dosáhli nárůstu rychlosti shlukování. Použití kd-stromu jakožto pokročilé datové struktury pro uchovávání objektů v k dimenzionálním prostoru nám velmi pomohlo s efektivním dotazováním do datasetu. Tato struktura sice potřebuje určitý čas pro vytvoření, pak je ale uložena v paměti a je schopna nám velmi rychle vracet výsledky. Jak jsme si ukázali v sekcích 4.2.2 a 4.2.3, rychlost a efektivita kd-stromu se snižuje nejen s rostoucím počtem bodů ale také velmi podstatně s narůstajícím počtem dimenzí. Tato vlastnost je ovšem vlastní všem datovým strukturám založeným na dělení prostoru (*space partitioning*) a je často nazývána anglickým termínem *curse of dimensionality*. K řešení problému ANN a NN v multidimenzionálních prostorech s počtem dimenzí přesahujících desítky dimenzi se nabízí například algoritmus *Locality-Sensitive Hashing (LSH)* [9]. Tento algoritmus pracuje na principu výpočtu hashe pro každý z bodů a vytvoření hash tabulky. Při dotazu na bod pak vypočte jeho hash a vrátí všechny body ze stejné části tabulky. Co se týče možností vyhledávání nejbližších sousedů, existuje několik způsobů jeho řešení. My jsme zvolili strukturu podporující vyhledávání k nejbližších sousedů, kterou jsme použili pro návrh našeho algoritmu. Mezi další přístupy patří například vyhledávání v určitém intervalu (*range searching*), které může být také velmi přínosné v řešení podobných problémů.

Z hlediska paralelního zpracování kódu se nabízí použití takové implementace kd-stromu, která bude spolehlivě pracovat i s paralelním přístupem k vyhledávání ve stromu. Tímto bychom se vyhnuli sekci `omp critical` a došlo by k dalšímu nárůstu rychlosti běhu. Jako další krok pro případné urychlení zpracování se nabízí možnost prozkoumat možnost paralelizace samotné konstrukce kd-stromu, ne pouze jeho dotazování. Pro tuto problematiku jsou dostupné návrhy algoritmů a postupů, například *Massively parallel KD-tree construction and nearest neighbor search algorithms* autorů L. Hu, S. Nooshabadi a M. Ahmadi [10]

Z hlediska paralelizace algoritmu a využití jiných technologií k paralelizaci samotné se i zde nabízejí různá řešení. Pomocí kombinace několika technologii (MPI a OpenMP bychom mohli úkol rozdělit na několik počítačů a na každém z nich zároveň využít lokální paralelizaci.

7.1 Jiná řešení

Pravděpodobně nejpokročilejší publikované řešení, které se věnuje problematice masivní paralelizace algoritmu DBSCAN je návrh nové varianty algoritmu s názvem *HPDBSCAN: highly parallel DBSCAN* [11]. Autoři M. Götz, Ch. Bodenstein a M. Riedel navrhli a implementovali variantu algoritmu, která využívá hybridní způsob paralelizace s použitím MPI i OpenMP. V počáteční fázi program rozdělí vstupní data pomocí tzv. *hypergrid*, tedy jakési mřížky v několika dimenzích a tato data distribuuje na jednotlivá vzdálená výpočetní místa pomocí MPI rozhraní. Každá z těchto buněk přesahuje do *epsilon* vzdálenosti vedlejší buňky. Následný výpočet lokálního DBSCAN algoritmu probíhá paralelně pomocí OpenMP. Princip sjednocení napříč buňkami pak spočívá ve vygenerování pravidel obsahujících data o totožných shlucích, která

jsou opět distribuována mezi výpočetními centry. Tímto způsobem byli autoři schopni zpracovat i velmi obsáhlé datasety o velikosti desítek milionů záznamu s rychlostním nárůstem v řádu tisíců procent na stovkách procesorových jader.

Podobný postup při paralelizaci DBSCANu prezentují autoři M. Ali Patwary a spol. [12], kteří přistupují k problému rozdělením datasetu, který rozbíjí klasický sekvenční průchod daty v sekvenční verzi algoritmu. K sestavení shluků používají princip stromu, kdy postupují odspodu nahoru a skládají shluky dohromady. Díky tomuto přístupu prezentují lepší rozložení paralelních výpočtů mezi jednotlivé výpočetní části. Algoritmus implementují pro sdílenou i distribuovanou paměť. Výsledkem je zrychlení až do hodnoty 5765x při použití 8192 jader na distribuované výpočetní architektuře.

Jiné řešení založené na přístupu master-slave prezentují X. Xiaowei, J. JÄGER a HP. KRIEGEL [13]. V jejich přístupu převádí sekvenční vyhledávání nejbližších bodů do paralelní podoby za pomocí distribuované verze R-stromu. Pomocí master-slave přístupu, kdy master uzel vytvoří a uchovává index a jednotlivé části dat jsou rozděleny slave uzlům. Následně po provedení výpočtu opět slouží shluky dohromady podle hraničních oblastí jednotlivých regionů.

8 Závěr

V této práci jsme si nejprve popsali teoretický koncept shlukování a zaměřili se speciálně na shlukování na základě hustoty. Algoritmus DBSCAN jsme si, jakožto nejpoužívanějšího reprezentanta této skupiny shlukovacích algoritmů, rozebrali a podívali se na jeho vlastnosti. Dále jsme si popsali výhody a možnosti použití pokročilých datových struktur, které umožňují efektivnější dotazování nad daty v prostoru. Z této oblasti jsme se věnovali nejvíce struktuře kd-strom, která je nezávislá na počtu dimenzí v prostoru a umožňuje velmi rychlé dotazování na nejbližšího souseda.

Pro vlastní řešení jsme se rozhodli použít kd-strom jakožto datovou strukturu pro ukládání bodů a jejich efektivní dotazování. V kombinací s touto strukturou jsme navrhli vlastní modifikaci algoritmu DBSCAN, kterou bude možné paralelizovat. Po porovnání jednotlivých přístupů k paralelnímu programování jsme rozhodli, že naše řešení založíme na technologii OpenMP a využijeme tak výpočtu na jednom počítači se sdílenou pamětí. Paralelizace algoritmu DBSCAN, který je velmi závislý na sekvenčním průchodu body se ukázala být nelehkou záležitostí. Podařilo se nám upravit algoritmus tak, že všechny jeho kroky lze paralelizovat a zrychlit tak jeho běh.

Výsledný efekt paralelizace ovlivnily dvě skutečnosti. Použitá implementace datové struktury kd-strom neumožňuje paralelizaci, a tak jsme byly nuceni tuto část kódu obalit direktivou, která zaručovala její bezpečný běh v paralelní části programu. Efekt tohoto řešení se nám následně prokázal v měření, kde jsme zjistili, že tímto krokem jsme omezili část paralelních bloků a snížili efekt více vláknového zpracování. I s tímto jsme ale byly schopni dosáhnou zvýšení efektivity běhu programu řádově o několik desítek procent. Ukazuje se, že paralelizace se projevuje nejvíce při použití dvou vláken oproti jednomu, efekt pak dále pokračuje při použití třetího vlákna. Další počet vláken už nám zrychlení programu nepřinese, neboť se režijní náklady vyrovnají se zrychlením výpočtu. Co se týče naší modifikace algoritmu DBSCAN, zjistili jsme, že díky jeho úpravě jsme schopni jej paralelizovat, ale stinnou stránkou zůstává nutnost navštívení každého bodu několikrát, hlavně ve druhém kroku programu, kde dochází k postupnému slévání vytvořených shluků. A právě tato vlastnost je druhou věcí ovlivňující běh programu. Námi modifikovaný DBSCAN umožňuje paralelní zpracování, většinu času jeho běhu ale algoritmus spotřebuje na tvorbu shluků ve druhém kroku své práce. Právě protože jsme potřebovali pracovat v několika vláknech nezávisle na sobě, museli jsme průchod daty rozdělit do několika částí a věnovat se vždy pouze jednomu bodu a jeho okolí. Tímto přístupem začínáme shlukování vždy s velkým počtem malých shluků, které následně slučujeme dohromady. A právě tato část nám zabírá nejvíce času. Je to ovšem také část, kterou jsme schopni díky paralelnímu přístupu nejvíce zrychlit.

Po změření výsledků jsme si popsali další možná vylepšení našeho řešení, například použití takové implementace kd-stromu, která umožňuje paralelní přístup.

V této práci jsme tedy popsali a implementovali paralelní verzi algoritmu DBSCAN s využitím pokročilé datové struktury kd-strom. Experimentálně jsme ověřili, že použití kd-stromu

i paralelizace algoritmu přinesly zrychlení běhu programu.

Literatura

[1] Hierarchické aglomerativní shlukování. In: *Matematickabilogie* [online]. 2008 [cit. 2018-02-02]. Dostupné z: `https://bit.ly/2GXR6BE`

[2] Hierarchické divizivní shlukování. In: *Matematickabilogie* [online]. 2008 [cit. 2018-02-02]. Dostupné z: `https://bit.ly/2IRQp9a`

[3] Clustering. *Scikit-learn: machine learning in Python* [online]. [cit. 2018-03-12]. Dostupné z: `http://ogrisel.github.io/scikit-learn.org/sklearn-tutorial/modules/clustering.html`

[4] The Silhouette Coefficient In: *cs.fit.edu* [online]. 2008 [cit. 2018-02-18]. Dostupné z: `https://cs.fit.edu/~pkc/classes/ml-internet/silhouette.pdf`

[5] BROWN, Russell A. Building a Balanced k-d Tree in O(kn log n) Time. *Journal of Computer Graphics Techniques (JCGT)* [online]. 2015, **4**(1), 50-68 [cit. 2018-03-14]. Dostupné z: `http://jcgt.org/published/0004/01/03/`

[6] MOUNT, David a Sunil ARYA. ANN: A Library for Approximate Nearest Neighbor Searching. *UNIVERSITY OF MARYLAND* [online]. Jan 27, 2010 [cit. 2018-04-14]. Dostupné z: `https://www.cs.umd.edu/~mount/ANN/`

[7] SUSY Data Set. *UC Irvine Machine Learning Repository.* [online]. [cit. 2018-04-21]. Dostupné z: `https://archive.ics.uci.edu/ml/datasets/SUSY#`

[8] *ALGLIB User Guide: Nearest neighbor search with kd-trees* [online]. In: [cit. 2018-04-14]. Dostupné z: `http://www.alglib.net/other/nearestneighbors.php`

[9] ANDONI, Alexandr. LSH Algorithm and Implementation (E2LSH). In: *Massachusetts Institute of Technology* [online]. 2008 [cit. 2018-04-14]. Dostupné z: `http://www.mit.edu/~andoni/LSH/`

[10] HU, Linjia, Saeid NOOSHABADI a Majid AHMADI. *Massively parallel KD-tree construction and nearest neighbor search algorithms.* 2015. DOI: 10.1109/ISCAS.2015.7169256. ISBN 10.1109/ISCAS.2015.7169256. Dostupné také z: `http://ieeexplore.ieee.org/document/7169256/`

[11] GÖTZ, Markus, Christian BODENSTEIN a Morris RIEDEL. HPDBSCAN. *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments - MLHPC '15* New York, New York, USA: ACM Press, 2015, 2015, 1-10. DOI: 10.1145/2834892.2834894. ISBN 9781450340069. Dostupné také z: `https://www.researchgate.net/publication/301463871_HPDBSCAN_highly_parallel_DBSCAN`

[12] HENDRIX, William, Md. Mostofa ALI PATWARY, Ankit AGRAWAL, Wei-keng LIAO a Alok CHOUDHARY. Parallel hierarchical clustering on shared memory platforms. *2012 19th International Conference on High Performance Computing.* IEEE, 2012, 2012, 1-9. DOI: 10.1109/HiPC.2012.6507511. ISBN 978-1-4673-2371-0. Dostupné také z: `http://ieeexplore.ieee.org/document/6507511/`

[13] XU, Xiaowei, Jochen JÄGER a Hans-Peter KRIEGEL. *A Fast Parallel Clustering Algorithm for Large Spatial Databases.* DOI: 10.1023/A:1009884809343. ISBN 10.1023/A:1009884809343. Dostupné také z: `http://link.springer.com/10.1023/A:1009884809343`

A Tabulka s naměřenými údaji

Tabulka 6: Vliv počtu vláken na délku běhu algoritmu.

Bodů a dimenzí:	1 vlákno [s]	2 vlákna [s]	3 vlákna [s]	4 vlákna [s]	`minPts`	`eps`	shluků	konstrukce kd-stromu [s]
2,5k 8D	1,01	0,62	0,51	0,48	7	1,0	13	0,13
5k 8D	2,53	1,7	1,68	1,63	7	0,9	19	0,27
10k 9D	7,1	4,76	4,32	4,22	7	1,0	90	0,97
20k 9D	7,35	5,96	5,6	5,52	10	1,0	55	1,94
50k 9D	66,54	49,71	48,52	48,14	8	0,9	312	4,91
100k 9D	152,27	115,69	116,35	117,25	10	0,9	338	9,91
250k 9D	606,29	480,21	485,44	492,68	14	0,9	496	25,42
50k 19D	128,87	117,14	117,85	118,21	10	1,0	40	10,22
100k 19D	345,12	318,14	325,41	331,85	10	1,0	123	21,12

Hodnoty jsou průměrem 5 běhů algoritmu pro každou konfiguraci.